LEVEL III

Bolt Beranek and Newman Inc.

Report No. 4066

Development of a Voice Funnel System: Design Report

R. Rettberg, C. Wyman, D. Hunt, M. Hoffman, P. Carvey, B. Hyde, W. Clark, and M. Kraley

August 1979

Prepared for:
Defense Advanced Research Projects Agency

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER (9)

4. TITLE (and Subtitle) (6)

DEVELOPMENT OF A VOICE FUNNEL SYSTEM: DESIGN REPORT.

5. TYPE OF REPORT & PERIOD COVERED

Design Report. for 15 Jul 78 - 31 Jul 79

6. PERFORMING ORG. REPORT NUMBER

4098

7. AUTHOR(s) (10)

R. Rettberg, C. Wyman, D. Hunt, M. Hoffman, P. Carvey, B. Hyde, M. Clark, and M. Kraley

8. CONTRACT OR GRANT NUMBER(s) (15)

MDA903-78-C-0356
ARPA Order-3653

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Bolt Beranek and Newman Inc.
50 Moulton Street, Cambridge, MA 02183

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

ARPA Order No. 3653

11. CONTROLLING OFFICE NAME AND ADDRESS (11)

Defense Advanced Research Projects Agency
1400 Wilson Blvd., Arlington, VA 22209

12. REPORT DATE

August 1979

13. NUMBER OF PAGES

252

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 294

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Unlimited

(14) BBN-4098

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents the current design of the Voice Funnel System, a high speed concentrator for packet switched speech and other streams of packetized data. The design is based on a multiprocessor architecture implemented using a novel switching network, the Butterfly Switch.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Report No. 4098 Bolt Beranek and Newman Inc.

DEVELOPMENT OF A VOICE FUNNEL SYSTEM: DESIGN REPORT

R. Rettberg, C. Wyman, D. Hunt, M. Hoffman, P. Carvey, B. Hyde, W. Clark, and M. Kraley

August 1979

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 30 November 1980
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Deputy Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

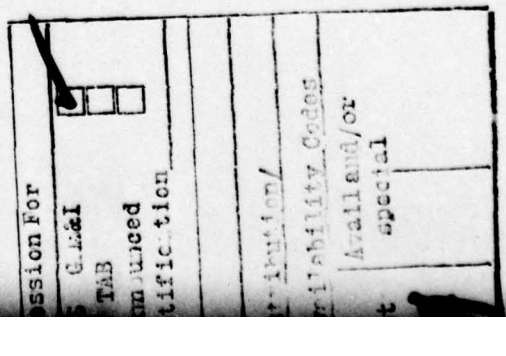


Table of Contents

Chapter I: Overview

Chapter II: The Voice Funnel Application

Chapter III: The Butterfly Switch

Chapter IV: Processor Node

Chapter V: System Software

Chapter I: Overview

Chapter I: Overview

Contents

Chapter I: Overview

1. Introduction

This document presents our current thinking on the design of a Voice Funnel System. It represents work performed during the first phase of the project, leaving approximately 20 months before delivery of the first two Voice Funnels. During this phase we have reviewed and updated our earlier ideas concerning the design of the Butterfly Switch and we have carried the switch design to a finer level of detail. We have also begun the design of the processor node and have performed the top level design and analysis of the application software and the operating system. Naturally, at this stage not all areas of the design are equally detailed, and we expect to modify it somewhat as our work progresses. Nevertheless, we believe that we have considered all major areas of the system at least at a high level and that we have a coherent overall design.

In addition to the Overview, this report consists of four chapters describing the four major aspects of the project. Because the primary goal of the project is to build a Voice Funnel, we first describe, in Chapter II, the requirements and characteristics of a Voice Funnel, together with our analysis of the major problems to be solved and the approaches we plan to take. Chapter III is concerned with the secondary goal of the

project, the construction of a Voice Funnel from a multiprocessor based on a Butterfly Switch. Chapter IV presents our design for the individual processor nodes, based on Z8000 microprocessors. Finally, Chapter V presents the operating system which we will build to organize the resources of the Butterfly Multiprocessor so that they may be easily and efficiently used by the Voice Funnel application software.

Anticipating that both the Voice Funnel and the Butterfly Multiprocessor will be successful, making eventual enhancement of both a likely possibility, we have been careful to include in our design the potential for expansion upon our early work. Thus in many cases the report suggests software features which may not be included in the initial version. While we do not expect to be able to do all of the things mentioned in the report, we are confident that we can produce a very useful and interesting initial system. Looking beyond the initial system, we feel that it is important to include in the report more than we might implement initially, because this will provide others with an early indication of our thinking and will increase our own assurance that our design is sufficiently robust to permit extension.

The rest of this chapter provides four brief overviews of the four main chapters of this report. The reader interested simply in knowing the direction of this work and not the details can stop after reading the rest of Chapter I.

2. Voice Funnel Software

The task of the Voice Funnel will be to interface to a large number of speech terminals which generate INTERNET packets. It will multiplex their data streams into a wideband satellite channel and demultiplex these streams for delivery to speech terminals at other sites. The initial Voice Funnel will support 1.5 Mbps of total traffic, although it is expandable to permit operation at higher speeds later.

The Voice Funnel is particularly well suited to hosts which generate traffic in the form of streams and make use of stream oriented internetworking protocols. Since the interface to the Voice Funnel is INTERNET packets, any host generating INTERNET packets may be connected to the Voice Funnel.

The functions of the Voice Funnel divide into two categories, main line tasks and background tasks. Main line tasks involve the actual handling of speech packets, and must be performed quickly enough so that the total delay of a speech packet in the Voice Funnel is kept sufficiently small. Background tasks are not directly involved with the transmission of individual packets; as a result, there is no firm latency requirement.

Speech data is received by the Voice Funnel as INTERNET packets, which include NVCP (or NVP-II) packets within them.

Each such packet includes digitized speech in the form of a small number of parcels or a stream of bits. In order to interface simple encoders which transmit only parcels or bit streams, a Voice Funnel option would be able to convert between simple speech data streams and INTERNET packets.

The Voice Funnel can be viewed as a network, with a central network switch in the Voice Funnel and network hosts implemented both within the Voice Funnel software and by external devices. External hosts include speech terminals. Internal hosts include a gateway to the PSATNET and the hosts which interface to simple encoders. The use of the network structure allows implementation of functional modules as hosts which may be attached or removed from the network as desired.

The main line functions of the Voice Funnel will be performed by a group of modules which will accept packets from speech terminals, aggregate them for transmission over the PSATNET (a wideband satellite network), segregate them at the other end, and deliver them to their respective destination speech terminals. The Voice Funnel will support speech terminals which are connected directly or over a network. Eventually it will interface speech terminals which transmit packets, parcels, bytes, or a continuous bit stream. Where needed, device dependent modules will convert between the format used at the interface and INTERNET packets. Other modules will take packets

destined for the satellite channel and merge them in stages into a large message buffer. Finally, a satellite network header will be attached as a prefix and the buffer will be transmitted.

Incoming messages will be broken up into individual INTERNET packets, and the packets will be passed to modules handling individual speech terminals or encoders. Where reconstitution is not provided by the speech terminals, these modules will perform speech reconstitution, a procedure by which the speech in the incoming packets will be restored to its original order, missing speech will be replaced by silence or by extrapolated sounds, and the resulting speech transmitted at regular intervals to restore smooth speech. Other modules will operate in the background. Background tasks will include call and conference establishment and control, and accounting.

An external controller may be provided outside the Voice Funnel in order to provide control functions such as call placement, conference chairing, and accounting, as well as experiment monitoring and control. In the absence of the external controller, an internal default controlled will provide simple versions of these functions. Use of alternate controllers will be simple since the controllers are implemented as hosts on the network.

The goals of low delay and high bandwidth have influenced much of the design of the application software. The Voice Funnel will employ several techniques for achieving high bandwidth in the Voice Funnel and over the satellite channel, including: multiplexing messages on stream traffic; minimizing copying of data and using the block transfer mode of the switch interface when appropriate; and increasing parallelism by using many small processes rather than a few large processes.

3. Butterfly Switch

The Voice Funnel System will be implemented on a newly designed machine, the "Butterfly Multiprocessor". This machine will consist of from one to several hundred processing units called "processor nodes" which will be interconnected by a switch called the "Butterfly Switch".

The Butterfly Switch is a network of elements, called "switch nodes", which route messages between processors and remote memories. All of the switch nodes in a Butterfly Switch are identical. Each node routes messages which appear on its input ports to one of several output ports on the basis of address bits at the beginning of the message. The route a message takes through the switch is determined by the interconnection pattern of the switch nodes.

A message is the basic mechanism of data transfer to or from remote memory in the Butterfly Multiprocessor. Each processor reference to non-local memory results in a request message traveling across the switch to the remote processor node, and a reply message returning across the switch. Bulk transfers between memories on separate processor nodes are accomplished by another class of messages. These and other messages traverse the Butterfly Switch.

A preliminary description of the Butterfly Switch has been presented previously [KRAL 78]. Efforts to further refine the design of the switch and understand its performance have led to several important improvements, which are presented in Chapter 3 of this document. The following points summarize these design refinements:

- The number of inputs and outputs on one switch node has been increased from two to four, significantly reducing the number of switch nodes in a Butterfly Switch.
- The data paths through the switch are four bits wide to increase the bandwidth of the switch and reduce the delay experienced by a message as it travels through the switch.
- The "retreat" strategy, rather than the "wait" strategy, will be applied when conflicts occur in the switch.
- Each message will carry a checksum which can detect errors in the data or routing of the message.
- An extra column will be added to large switches in order to minimize the effect of a failure in a switch node.

Increasing the number of input and output ports on a switch node from two to four reduces the number of switch nodes by a factor of four. We use the term "base" to denote the number of input or output ports on a switch node. The table below gives the number of nodes required to implement switches with base-4 and base-2 nodes.

	Number of Switch Nodes	
Ports	Base-4	Base-2
4	1	4
16	8	32
64	48	192
256	256	1024

Base-4 nodes also reduce the number of columns in a switch and thus the delay through the switch and the likelihood of contention. The primary disadvantage of base-4 switches is that they grow less gracefully than base-2 switches. Each base-2 switch has twice as many ports as the next smaller size switch while each base-4 switch has four times as many ports as the next smaller switch. This makes it harder to match the desired number of ports to the size of the switch.

We have discovered that the simple "wait" strategy for resolving conflicts is prone to deadlocks. Substituting the "retreat" strategy eliminates the deadlock and permits a switch interface design which has no known deadlocks.

Two mechanisms will make the switch more robust in the event of switch node failures: checksums and extra columns. Each message will carry a 4-bit checksum to help detect errors in its data, and the destination processor node address will be included in the checksum to detect errors in message routing.

A Butterfly Switch is particularly vulnerable to the failure of a switch node which is on the interior of the switch since such a failure may isolate many processor nodes. An extra column of switch nodes provides an extra path through the switch which may be used to route messages around the faulty switch node. Since a 16 input port switch has only two columns, there are no

"interior" columns and adding an extra column would not make sense. As a result, extra columns will be added only to larger switches.

Simulations of the Butterfly Switch have now been performed which show that the effect of references to global memory becomes important as the number of global memory references approaches 5% of the total. The expected global memory reference rate for the Voice Funnel is unknown at this time. However, by executing instructions from local memory and requiring that the stack and private variables also be local, we will be able to reduce the global reference rate significantly below what it would be if we ignored locality entirely.

In order to calibrate the effect of the Butterfly design upon the switch, we also simulated a crossbar switch having the same data path width. Interestingly, the simulations also show that the performance of a Butterfly Switch does not differ greatly from that of a crossbar switch of equal path width.

An MSI implementation of the switch node has been designed which requires between 40 and 60 MSI TTL packages. We expect that this design will yield a switch with a 48 Mbps potential data rate between any pair of input and output ports. For small machines, such as the prototype Voice Funnels, a switch node of this complexity could be implemented on a moderately small

printed circuit card. A large machine with, say, 256 processor nodes would have 256 switch nodes, making an LSI implementation of the switch node more attractive.

Up to now, we have assumed that request messages and reply messages would travel through the switch independently. As an alternative, we might design the switch so that the path from the source to the destination would be held after the request message has been sent so that a reply could be returned over the same path. This approach could have several advantages:

- the source processor node address need not be sent, shortening the request message;
- the reply message would not suffer contention either within the switch or at the source processor node;
- the "wait" strategy would be free from deadlocks and could be employed if desired.

The "bidirectional" switch is more complicated than a "unidirectional" switch and may run slower. Holding the path for the reply would cause some switch paths to be occupied while no data was being sent. This would increase the number of conflicts in the switch. The balance between these advantages and disadvantages is unknown at this time.

4. Processor Node

Processing in the Butterfly Multiprocessor will be performed by modules called "processor nodes". A processor node will consist of a processor, I/O devices, memory, and an interface to the Butterfly Switch. The current design permits Butterfly Multiprocessors which contain up to several hundred processor nodes.

The processor used in the processor node will be Zilog's Z8000. The Z8000 is a highly integrated microprocessor which represents the state of the art in commercial microprocessor design. Several characteristics make the Z8000 particularly attractive:

- the "segmented" version supports 23-bit virtual addresses;
- a compatible memory management unit has been announced; and
- the Z8000 is a reasonably fast 16-bit processor.

The ability of the Z8000 to support a virtual address space of greater than 16 bits is particularly important. Multiprocessors are often large machines and are expected to handle large problems. These large problems require large programs manipulating large data spaces to solve them. A large virtual address space is necessary to accommodate these memory addressing requirements. In current machines, several techniques such as base registers and mapping registers have been used to

expand a 16-bit address into a larger address. These solutions miss the point, however. In a large data structure, pointers to objects must be atomic units in order to eliminate translations when using a pointer. The most straightforward way to provide this is to support longer virtual addresses at the outset. The Z8000 permits manipulation of 23-bit virtual addresses and 23-bit pointers embedded in 32-bit double word operands. Eventually even larger address spaces may be required, but by that time we expect microprocessors with larger virtual address spaces to be available.

The previous discussion has concentrated on the address space seen by the programmer (or compiler). The hardware supports another address space -- the physical address space. During each memory access, the virtual address is translated into a physical address by a device called a Memory Management Unit. The Memory Management Unit uses a translation table maintained by the operating system. Memory management facilitates protection by limiting a process to only those areas of physical memory which are specified in the translation table and by allowing each process to access memory only as specified by protection flags in the translation table.

A benchmark program has been written for the Z8000 and has been used to compare the Z8000 with the the LSI-11. These results are based on preliminary specifications of the Z8000 and

should be regarded as preliminary themselves. Briefly, we have found that:

- The code density of the Z8000 is equivalent to that of a PDP-11;
- It is straightforward to code those things that programs do most often;
- A few nonuniformities in the Z8000's instruction set might make code generation slightly more difficult than for the PDP-11;
- The Z8000 in the segmented operating mode is about 2.6 times as fast as an LSI-11;
- The average instruction execution time is about 2 microseconds.

Commercial microprocessors are not designed for use in tightly coupled multiprocessor systems. As a result, one must expect some incompatibilities between the processor and the rest of the processor node design. For example, the Z8000 expects that all of its memory is "close" to the processor. As a result, it seizes the bus for the duration of a memory access. In a Butterfly Multiprocessor, some accesses will be made to memory which is in the "global" address space. Were we to permit the Z8000 to hold the bus while the access is in progress, important real time events might be blocked. Other examples include the method of refreshing main memory and timing considerations. In each of these cases, specialized circuitry is required in order to correct for these deficiencies.

Several I/O devices are common to every processor node:

- The memory management unit described above;
- A counter-timer which will be used to provide interrupts for real-time or scheduling events;
- A Read Only Memory which is used for system loading and debugging;
- A serial interface for loading, system control, and debugging.

In addition to these standard I/O devices, the Butterfly Multiprocessor will require interfaces to support the application. To the extent possible, these interfaces will be implemented using standard LSI interface circuits such as USARTS or advanced communications protocol chips. Where bandwidth considerations are important, these interfaces will be direct memory access devices.

Each processor node will contain an interface to the Butterfly Switch. This interface will offer several communications services. It will identify memory accesses which are to global memory and convert them into messages addressed to the correct processor node. It will also respond to request messages from other processor nodes for accesses to its own local memory. The programmer will be unaware when an access uses the switch.

The switch interface will also be able to transfer a block of words between the memory in its own processor node and that in

another processor node. This type of transfer must be specified explicitly by the programmer, but it will make very efficient use of the bandwidth of the switch. To use this mode, the programmer specifies the direction of the transfer, the starting virtual addresses of both the local and remote memory regions, and the number of words to be transferred. The switch interface will perform the transfer, making up request messages and dealing with replies. In order to reduce the impact of long messages on other users of the switch, long transfers will be broken into blocks of at most 8 or 16 words.

In addition to data transfers across the switch, the switch interface will support special messages which perform functions such as interrupting a remote processor, amputating the processor, or locking a resource.

We expect that a processor node will occupy two large printed circuit cards, one for the I/O devices, and the other for the processor, standard I/O devices, memory and switch interface.

5. System Software

Supporting the Voice Funnel application there will be an operating system which will provide the following services:

- processor management (or scheduling),
- memory management,
- inter-processor communication,
- process synchronization,
- interrupt handling,
- system configuration, and
- reliability assurance.

The operating system will provide an environment in which the work of the Voice Funnel application can be partitioned into a number of individual tasks which can be performed in parallel by several processes running simultaneously on different processors. In our model, tasks and processes are parallel concepts: a task is the unit of work performed by a process (although a process which cycles may handle a series of tasks).

Because the Butterfly Switch will permit all processors to access all available memory directly, cooperation between processes on different processors will be almost as easy as between processes on the same processor. Not only will the operating system take advantage of this feature itself, but it will also provide facilities to make it safe and easy for

application processes to communicate freely regardless of which processor they are running on. While accesses to remote memory may be convenient, however, they will encounter both delay and contention, and excessive use of remote memory could seriously degrade overall system performance.

In order to efficiently use the available hardware while still achieving high parallelism and low delay, the application software and the operating system will observe certain conventions:

- a process may only run on the processor in whose local memory it resides; in consequence, processes which might run on several processors must be replicated;
- a task, however, may be performed by any copy of the process written to handle it, regardless of the processor the process runs on, except when a specific process is required for some reason;
- in order to minimize delay for individual tasks, processes which will perform frequently occurring tasks will be created in anticipation of their use (on one or more processors) and will remain resident;
- except under unusual circumstances, processes will not be moved from one processor to another; normally, when a process must be run on another processor, a copy of the process will be created there.

Scheduling in the Voice Funnel must provide a mechanism for efficiently using many processes and processors to perform a large number of tasks with low delay. Because tasks can often be performed by any copy of the process which handles them, it is possible to take advantage of multiple processors by having a

global scheduling function which directs incoming tasks to suitable processes on processors which are not too busy. Because processes will be created in advance of their use and can only be run on the processors to which they are local, global scheduling of processes will not be worthwhile. However, local scheduling will be necessary in order to allocate time on a priority basis among the processes assigned to each processor.

Three potential global scheduling algorithms are discussed in the report:

- an implicit algorithm having no explicit global scheduling module, but instead relying upon natural tendencies in the system to cause tasks to flow toward processors which are less busy;
- a decentralized, but explicit, global scheduler which would pull tasks toward processors as capacity became available to them; and
- a centralized, but migrating, global scheduler which would push tasks towards the processors with the most excess capacity.

The implicit algorithm will be implemented first because of its simplicity and naturalness. Its performance characteristics will then be observed, and one of the other algorithms will be considered if the implicit algorithm does poorly in practice.

Each process will have access to a large virtual memory (up to 8 megabytes on the Z8000). Because the memory will be segmented (up to 127 segments on the Z8000), and because any process can directly access memory in any processor, it will be

possible for processes to selectively share memory with one another in order to collaborate on a problem or to communicate rapidly, while concealing information such as instructions or private data from processes which should not be able to read or modify it. If not used carefully, these memory access and protection features could add significant overhead through memory fragmentation, context switching, or translation of shared references. Thus, virtual address spaces (with their size, protection, and sharing attributes) will be specified at link time, rather than at run time, retaining most of the flexibility offered by the memory management hardware without incurring significant overhead.

Reliability mechanisms similar to those found in the Pluribus Multiprocessor will be provided in order to exploit the reliability potential inherent in a multiprocessor. [ROBI 78] [KATS 78] Among the ideas which will be borrowed from the Pluribus are:

- discovery and verification of the hardware configuration at system initialization time;
- development and maintenance of a "consensus" regarding the set of correctly functioning common resources;
- periodic checking of critical resources;
- use of redundant information for verification of data structures;
- timeouts for locks on common resources; and

- blocking of higher level activities when lower level activities have detected possible errors or inconsistencies in resources upon which the higher level activities depend.

A UNIX time-sharing system will provide the basic environment for software development. A limited number of tools will be created (mostly by modifying existing tools) to permit development and maintenance of the application and the operating system to proceed rapidly. Both the application software and the operating system will be written primarily in the C Programming Language, although certain frequently executed or machine-dependent modules will be written in assembly language. The required tools will include:

C Compiler,
Assembler,
Linker,
Loader (runs on the Butterfly Multiprocessor),
Down Line Controller,
Debugger, and
Various Text Processing Programs.

6. References

[KATS 78] D. Katsuki, E.S. Elsam, W.F. Mann, E.S. Roberts, J.G. Robinson, F.S. Skowronski, and E.W. Wolf, "Pluribus -- An Operational Fault-Tolerant Multiprocessor," Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

[KRAL 78] M.F. Kraley and R.D. Rettberg, "A New Multiprocessor Architecture," BBN Report No. 3501, Reprinted December 1978.

[ROBI 78] J.G. Robinson and E.S. Roberts, "Software Fault-Tolerance in the Pluribus," AFIPS Conference Proceedings 47, 1978.

Chapter II: The Voice Funnel Application

Chapter II: The Voice Funnel Application

Contents

Chapter II: The Voice Funnel Application

1. Introduction

This chapter will examine the job to be done by the Voice Funnel, including the requirements of the job, strategies to be employed, and goals to be achieved. In the succeeding chapters we will look at the Butterfly Multiprocessor and the support it will offer for the Voice Funnel application. The hardware and operating system provide an environment which will support the the Voice Funnel application's requirements. The application software described in this chapter provides a communication network for packetized speech in the internet environment.

The primary task of the Voice Funnel is to interface a large number of speech terminals, multiplex their data streams for delivery over the Wideband Satellite Channel, and demultiplex these data streams for delivery to terminals at other sites. Speech packets consist of transport and speech protocol information and speech data. The transport protocol is expected to be INTERNET while the speech protocol is expected to be some version of NVP or NVCP.

In its role of transport and multiplexing, the Voice Funnel is not limited to the traffic from packetized speech encoders. Since the transport protocol being used is INTERNET, any other communications resource which uses INTERNET may also be attached to the Voice Funnel. Examples might include conventional hosts or new application hosts such as graphics terminals which want to take advantage of the high bandwidth available on the satellite channel.

Most of the speech terminals which will attach to the Voice Funnel will include sufficient intelligence to manage all speech processing tasks and generate complete addressing for fully packetized speech parcels. However, the Voice Funnel may include facilities to assist speech encoders which lack such intelligence by packetizing, addressing, and reconstituting speech parcels on their behalf.

2. Logical Structure of the Voice Funnel

The Voice Funnel is a component of an internetwork environment. It interfaces to a number of networks as well as to a large number of independently addressed hosts. It provides transport functions for these hosts according to addresses in each message. The hosts may be attached directly to the Voice Funnel, on local networks, on other Voice Funnels, or even on remote networks. As a result, the Voice Funnel may best be viewed as a network and an INTERNET gateway (Figure 2-1). Encoders may be attached as hosts on the Voice Funnel network, or through other networks, in which case their network is attached through the gateway. The satellite network is similarly attached through the gateway.

The gateway is perhaps the single most important host on the Voice Funnel network. It will switch traffic between any of the attached networks, including the Voice Funnel network. Note that traffic between an encoder on a local net and the satellite network will not pass through the Voice Funnel network switch, but will be switched by the gateway.

The heart of the Voice Funnel network is a switch implemented by the Voice Funnel software. This switch is the center of a star network. The hosts on the network fall into two categories, those implemented in software in the Voice Funnel

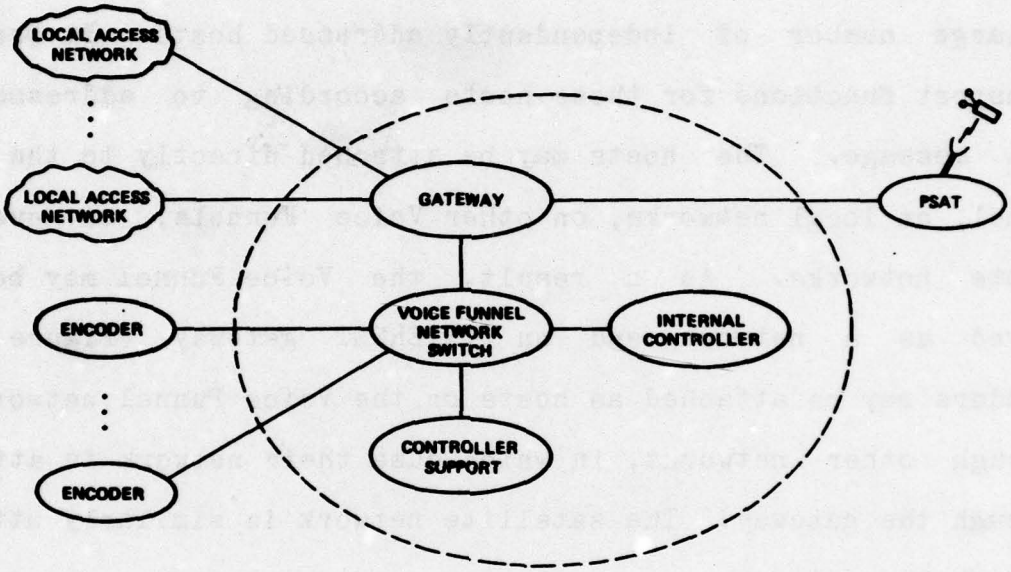


Figure 2-1 Structure of a Minimal Voice Funnel System

(e.g. gateway, internal controller) and those implemented as physically separate machines (e.g. speech terminals, external controller). A speech terminal on a Voice Funnel will use network services to communicate with another speech terminal directly connected to the same Voice Funnel, or, more generally, will use INTERNET services to communicate with a speech terminal attached through a local network, another Voice Funnel, or any other network.

In the initial implementation we expect much of the traffic to be between speech terminals either directly connected to Voice Funnels or attached through local networks, with the Voice Funnels communicating over the PSATNET. Such traffic would involve the traversal of three networks: the originating local network or Voice Funnel network, the PSATNET, and the destination local network or Voice Funnel network. Gateways in Voice Funnels at each end of the PSATNET would route the messages across the internet.

The minimal structure for the Voice Funnel will involve the network switch, interfaces for speech terminals, and two internal hosts -- the gateway to the PSATNET and local access networks and a controller support module. The controller support module is a host which provides the necessary internal "hooks" to allow a controller to sense and manipulate the state of the Voice Funnel. A number of options for expansion of the Voice Funnel are shown in Figure 2-2. These options for enhancement beyond the minimal Voice Funnel system include:

-- Expansion of the gateway to include interfaces to other networks;

-- An external controller which would provide services similar to but more extensive than those provided by the internal controller;

-- An internal host to provide conference control and chairman services as specified in the NVCP protocol;

-- Internal hosts which provide NVCP and INTERNET protocol services for simple encoders which provide only data

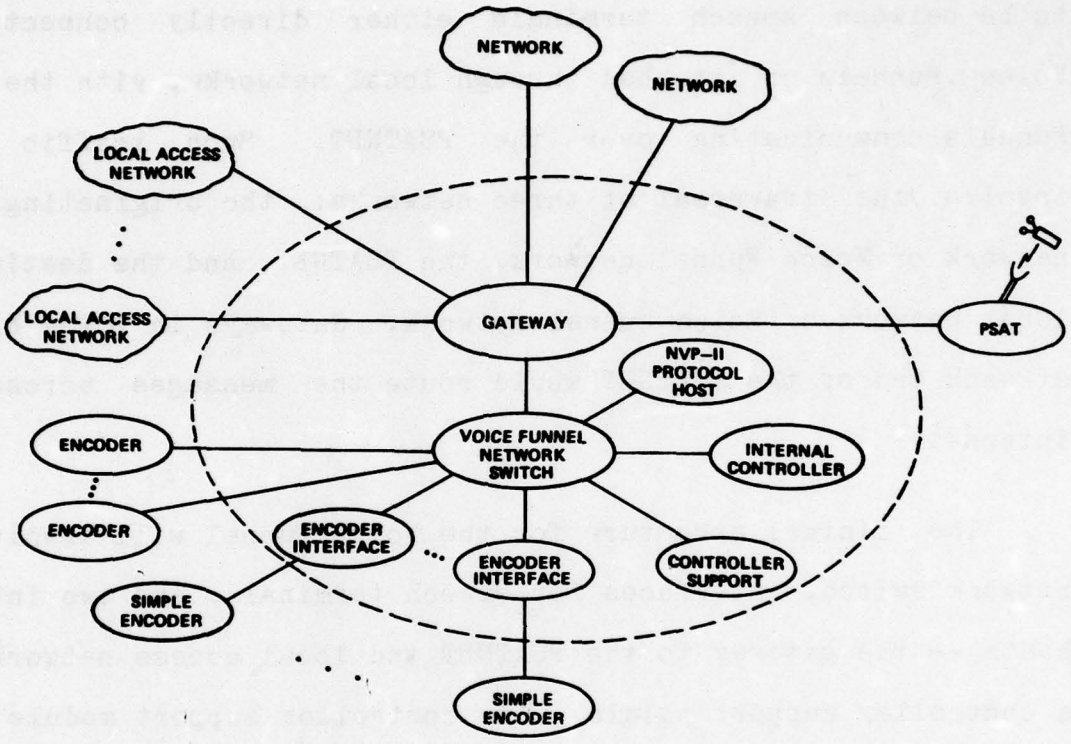


Figure 2-2 Structure of a Voice Funnel System With Options

streams or parcels and cannot function as network hosts directly;

-- Internal hosts to interface non-speech devices such as facsimile machines which might usefully employ the Voice Funnel's high bandwidth transmission capabilities.

By providing the basic structure of a central network switch with both internal and external hosts, these and other facilities may be added either inside the Butterfly Multiprocessor or externally

without disturbing the structure of the Voice Funnel or compromising the integrity of its basic services.

3. Speech Terminal Interface

Speech terminals interface to the Voice Funnel as hosts on a network. A network host interface requires protocols for electrical interface, for framing, and for transport. In addition, these interfaces often contain protocols for flow and error control as well as host and network status. We expect to support a number of protocols at the electrical and framing levels; for transport protocol we will accept only INTERNET. INTERNET protocol will be used since the main speech traffic is expected to be among encoders on distinct Voice Funnels. NVCP will be used by the encoders to support the specific requirement of the speech application. The Voice Funnel need not support NVCP since it is a protocol between speech terminals which is transparent to the Voice Funnel.

Flow and error control protocols may be necessary where the network provides (and the hosts assume) reliable delivery of packets. In the PSATNET, as well as in the speech community, reliable packet delivery through error recovery mechanisms is not assumed and not even desired. Inasmuch as the protocol of the Voice Funnel is INTERNET, the hosts can use TCP to provide reliable transmission when it is required.

The encoders will operate in a range of 2.4 kbps to 64 kbps and some may be variable rate. Except for packet size and

frequency, the differences among encoders will be invisible to the Voice Funnel software.

We will provide an interface based on a USART chip such as the 2651 (asynchronous or synchronous) or 2652 (synchronous) which will provide a flexible but inexpensive high speed interface to speech terminals. These chips include such services as CRC checksums and automatic synchronization character generation. Later, we may add local area net interfaces which would require channel arbitration and addressing mechanisms, allowing many speech terminals to share a single channel, or permit a remote terminal concentrator to communicate with the Voice Funnel.

4. PSATNET Gateway

The gateway to the PSATNET will be an internal host whose function will be to pass packets between the Voice Funnel and the PSAT for transmission across the PSATNET. It will receive packets from the Voice Funnel switch and from the PSAT, remove the local net header, and route the packet to its INTERNET destination over whichever network (PSAT or Voice Funnel) is appropriate.

Considerations of reliability, efficiency, and the special characteristics of the PSATNET bandwidth allocation scheme add to the demands on the gateway. Since the gateway is on an important data path for large amounts of traffic, prudence dictates that two interfaces to the PSAT be provided to permit reliable operation in the face of failures in the Voice Funnel-PSAT interface. Also, it is desirable to multiplex the PSAT messages to include many INTERNET packets. Finally, the necessity to preallocate stream resources in the PSATNET requires that the gateway be able to estimate or otherwise determine its bandwidth requirements over the PSATNET.

4.1 PSAT Interface

The PSATNET is the satellite network being developed as part of the ARPA wideband network project. The PSATNET will initially consist of four earth stations, each of which is connected to a

Pluribus Satellite IMP, or PSAT. Each Voice Funnel will be connected to a PSAT through two synchronous modem interfaces, each of which operates at rates up to 1.5 Mbps. Although either interface by itself should be able to be driven at 1.5 Mbps by the Voice Funnel, the two will operate in parallel and one will assume the whole load if the other fails. The processor nodes in the Butterfly Multiprocessor that contain these interfaces are points of particularly high bandwidth requirements and may receive special attention, as will be described later. The particular processor nodes in the Butterfly Multiprocessor that connect to the PSAT will send and receive information across that channel using DMA transfers.

The physical characteristics of the interface between the Voice Funnel and the PSAT computer have not been fully determined. We expect the channel to be synchronous with a bandwidth of at least 1.5 Mbps. We would like to use a standard link protocol with as much of the interface protocol as possible performed by a USART chip such as a 2651 or 2652, which can handle a data rate of 2 Mbps and provide CRC checksums and synchronization characters on the line.

4.1.1 PSATNET Stream and Datagram Service

The PSATNET provides two kinds of service: datagram service and stream service. If a message is sent over the PSATNET as a

datagram, then a reservation for satellite channel bandwidth is made on a per-message basis. In the case of a stream, it is necessary to reserve channel bandwidth only at stream initialization, or at times when expected load changes significantly. A stream reservation makes a fixed amount of channel bandwidth available on a periodic basis.

Both the datagram and the stream services provide for point-to-point and group addressing. It would be desirable for conference calls made through a Voice Funnel to make use of the PSATNET group addressing. However, much design remains before this can be achieved.

If a steady flow of information is to be sent over the PSATNET, it is more efficient to transmit it as stream messages rather than datagram messages, since the stream setup can be amortized over a number of messages. The Voice Funnel will use the PSATNET stream service for sending and receiving voice messages. By multiplexing together messages from a number of voice conversations, the Voice Funnel can statistically average the variation in the individual streams to produce digitized voice output that is less bursty than the output from a single encoder, and thus make particularly efficient use of the PSATNET stream service. A second reason for using stream rather than datagram service for voice traffic is to minimize delay: in stream service, it takes at least one satellite hop to send a

message, but in datagram service it takes at least two hops since a first hop is required to make the channel reservation.

The Voice Funnel will make some use of the PSATNET datagram service to send control messages. In particular, datagrams must be used between the Voice Funnel and the PSAT to set up new streams or to change stream characteristics. In addition, the datagram service may be used to send out-of-band control messages associated with streams. The high reliability that control messages typically require can be provided by the datagram service but not by the stream service, since acknowledgement and retransmission of messages is provided in the PSATNET only with datagram service. The use of datagram service does not eliminate the need for a reliable transmission protocol such as TCP in the Voice Funnel since the PSATNET datagram service provides neither duplicate detection nor ordering.

4.1.2 Use of the PSATNET Stream Service

The Voice Funnel will rely on the datagram service primarily for control messages and on the stream service primarily for voice data transport. This section describes the use of the stream service in greater detail, in order to show how the real time response requirements of the voice traffic can be met through the use of PSATNET streams.

For every stream, there is a predictable portion of the PSATNET channel bandwidth that is provided to stream users, since the overall channel bandwidth is time multiplexed to provide each stream. A stream consists of a sequence of stream slots. The stream slots for a given stream are all the same size, and they are periodic, i.e., the average elapsed time between stream slots is constant. The interval between the beginning of a stream slot and the beginning of the next stream slot is called the stream interval. A stream with stream slots that are 1 millisecond long and stream intervals that are 20 milliseconds long would use 5% of the available channel bandwidth.

Multiple PSATNET messages to a single host can be sent in the same stream slot. The use of group addresses permits a PSATNET message to be sent to different hosts as long as the destination hosts can be predetermined (such as in a conference). At this time it has not been determined to what extent multiple PSATNET messages to different hosts can be sent in the same slot. For the purpose of this report, we assume this latter capability. If it is not made available, this should have only a minor effect on the proposed design. In the initial implementation, each Voice Funnel will transmit messages in two streams: one for point-to-point traffic and one for conference traffic. Thus, there may be twice as many streams in use as there are Voice Funnels.

The use of separate streams for point-to-point and conference traffic arises from two opposing considerations: conservation of satellite bandwidth and minimization of overhead due to excess traffic in the Voice Funnels. If we used a single stream for all outgoing traffic from a Voice Funnel, we would use the satellite bandwidth most efficiently by maximizing the opportunity for statistical multiplexing. However, the stream would have to be addressed to all Voice Funnels which received any traffic, and each Voice Funnel would therefore have to sort out its intended traffic from all the voice traffic in the system.

At the other end of the spectrum, we could target traffic exactly by having one outgoing stream for point-to-point traffic in which messages would be sent to single Voice Funnels, and one stream for each conference. However, the conference traffic would not benefit from any multiplexing, since each conference is separate. By sending conference traffic to all hosts and point-to-point traffic to individual hosts, we hope to gain the advantage of considerable statistical multiplexing for all traffic, while still targeting point-to-point traffic to only the proper recipients. To provide separate stream addresses for each conference would, in addition, involve considerable stream setup overhead.

The contents of any stream slot can be sent from the Voice Funnel to the PSAT in one or more DMA transfers, as long as the overall timing constraints for that stream slot are observed. If no transfer is made during a stream slot time, then that particular portion of the PSATNET channel time is simply wasted.

There are other parameters, such as priority classes, that can be specified for streams. However, the stream slot and interval sizes are the two parameters that merit further study. The parameter values mentioned above, a 1 ms. slot size and 20 ms. interval size, are likely values in a lightly-loaded Voice Funnel. As the load increases, the slot size will grow while the interval size remains constant. Using a stream in this way makes it possible to deliver additional digitized speech to each receiving encoder every 20 milliseconds.

4.1.3 Use of Multiple PSAT Connections

Two links from the Voice Funnel to the PSAT are provided to spread the load and in case of failure of a single link. As part of the PSATNET protocol, short "hello" messages are sent periodically across each link between the Voice Funnel host and the PSAT, if there is no data traffic. If, in the absence of data traffic, either side does not receive a "hello" message within a specified timeout interval, the link is assumed to be inoperative and a restart protocol is invoked in an attempt to

bring the link back to an operational state. During the time that a link is not operational, all PSATNET messages that had been traversing that link will be re-routed over the other link.

The drivers of both PSAT links will use multiple buffers for both input and output in order to achieve maximum throughput. When both links are operational, output information from any given encoder will be moved to an output buffer for one of the two links. The output link chosen is arbitrary, and an attempt is made to balance the output traffic over the two links. Input from the PSAT is handled in the same way.

This approach expands to future versions of the Voice Funnel that connect to networks with very high data rates, such as 20 to 60 Mbps. In this case, the number of distinct links to the network that terminate at the Butterfly Multiprocessor would be much larger than two. Between the Voice Funnel and the network there may need to be a special-purpose device that only performs fan-in to the network and fan-out to the Voice Funnel. Even for this very high bandwidth case, the current PSAT input-output scheme should be able to scale up to support a large number of links since the association of speech data to PSAT links is arbitrary. (These rates, of course, are far in excess of the abilities of the PSAT.)

An upper bound on the throughput on a full duplex PSAT link that a given Butterfly Multiprocessor Processor Node can support can be estimated as follows. Each node in the Butterfly Multiprocessor can transfer one 16-bit word in or out of memory in 750 nanoseconds using the DMA interface. This is equivalent to 1.33 million words per second, or about 21.3 Mbps. If a node is doing PSAT input and output as well as input and output across the switch in the steady state, then essentially one-fourth of the total available bandwidth, or about 5.3 Mbps, is devoted to each of these activities. If half the references to the memory in that node come from the processor, then the maximum bandwidth is about 2.6 Mbps. A single node can support 1.5 Mbps full duplex even if more than two-thirds of the memory references are from the processor.

It is not necessary to have both the input and output sides of the interface attached to the same processor node. Also, at high data rates, it might be reasonable to dedicate the processor to support of the I/O device on an interrupt basis, with the processor idle between interrupt events. In that case, almost half of the 21.3 Mbps bandwidth would be available to the satellite interface -- a limit of 10 Mhz. We should note that these latter calculations are certainly optimistic and are not likely to be achieved in the current design. However, they illustrate the margin for error provided in this structure.

4.2 PSAT Scheduler

The PSAT Scheduler is responsible for the multiplexing and scheduling of PSATNET buffers. It manages all stream traffic across the Voice Funnel-PSATNET interface.

For outbound traffic the PSAT Scheduler aggregates the messages from all the processor nodes of the Voice Funnel into a PSAT buffer located on a node with an interface to the PSAT. When the buffer is full, or when delay and timing considerations require that the buffer be sent, the PSAT Scheduler passes the buffer to the interface device driver and thus to the PSAT. A suitable PSATNET header is affixed to the buffer.

The PSAT scheduler must also establish PSATNET streams and negotiate slot and interval sizes for each stream. This requires determination of expected traffic densities over each stream. Some techniques for determining traffic densities are discussed in section 9.1.2.

Inbound PSATNET traffic arrives at the scheduler from the interface device driver. The PSAT scheduler demultiplexes it and the messages within are routed to their destination. The PSAT scheduler must distribute the messages very quickly in order to (1) deliver the messages with minimal latency and (2) make the incoming PSATNET buffer available for more traffic as quickly as possible. Since these buffers are large (probably about 2000

bytes each), excessive delay would involve tying up a large amount of memory on the processor nodes with PSAT Interfaces; this memory could prove to be a critical resource.

There are two central considerations in the design of the PSAT scheduler: (1) it must merge messages from and distribute messages to all nodes of the Voice Funnel efficiently; and (2) it must work quickly so as to minimize message latency, keep up with the speed of the PSAT interface, and release large buffers quickly.

Models for the PSAT scheduler are being considered which would allow much of the activity to be performed in parallel. We expect that a single process implementation of the PSAT scheduler would not be able to keep up with the required data rates. Therefore, we are examining parallel algorithms for merging data into the PSAT buffers, and for sorting data from incoming buffers.

5. External Controller

The External Controller is an optional component which may be provided for the Voice Funnel. Its job is to perform those functions which are experimental in nature or of low bandwidth such as call setup or conference control, leaving the Voice Funnel to perform the high bandwidth tasks. There are arguments for making this controller a separate machine and even for developing it by some other group. Whether the controller is external or internal and implemented by BBN or some other group, it is valuable to structure the software so that the "main line" packet processing is separated from the low frequency tasks. This division preserves the speed and efficiency necessary in the main line while permitting a more convenient environment for development, maintenance, and experimentation with the controlling functions.

The External Controller may not be available at all times. In order to support traffic when it is not available, and to facilitate initial development and checkout of the Voice Funnel, a small internal software controller module will be provided to operate in the absence of the External Controller. It will provide essential services which the External Controller would otherwise provide so that the Voice Funnel will not be wholly dependent on the External Controller. In addition, some of the services of the controller may initially be duplicated by the

debugging system provided in the development environment (see Chapter 5).

The External Controller will be attached to the Voice Funnel as a host, using the same system of protocols used by other hosts such as the encoders. Since it is a host, the network interface protocol is INTERNET. Because the control information should be reliably transmitted between the machines even if the External Controller is not directly attached to the Voice Funnel, the TCP will probably be used as the host level protocol. In addition, the External Controller will include a protocol for communicating with the controller support module, an internal host which will carry out its commands. Thus, the controller is largely independent of the internal structure of the Voice Funnel, and alternative controllers may easily be substituted, including the switching between internal and external controllers as the availability of the External Controller changes dynamically.

5.1 Call Connection

The placement of calls and the establishment of conferences is outside the main line data path for the Voice Funnel and is therefore a candidate for service by the controller. Call placement involves determining the address of the receiver of the call, setting up of data paths for the speech packets and servicing channel capacity for the traffic flow.

Addressing may be a fairly simple function if the user provides the information in digital form, but it may also be provided in less direct terms. The user may provide the name of the receiver, from which the number must be determined from a table, or through an associative search. Experiments might be conducted in which the name or number of the receiver might be spoken into the encoder, and the controller would have to rely on voice recognition techniques to determine the correct receiver. Depending on the complexity of the algorithm for identifying the parties to the conversation, the Voice Funnel may need recourse to external assistance. The External Controller could provide an appropriate test bed for such techniques. The Voice Funnel would provide a less sophisticated form of addressing to the user in the absence of a connected controller (or until the controller is developed).

Data path set-up and channel reservation are difficult problems under the current protocols. The NVP and NVCP protocols specify call setup as a function strictly between speech terminals. Making the controller a party to call set up negotiation would enable the controller to influence encoder rates based on available bandwidth, and to know immediately and accurately the bandwidth needs of conversations. Alternatives are speech terminal to controller protocols involving simultaneous negotiation and monitoring of traffic by the

controller to estimate current bandwidth needs. Based on these negotiations or monitoring, the External Controller would request the PSAT scheduler to reserve, if possible, the bandwidth required for the conversation; in determining its needs, the External Controller would take into account the effects of statistical multiplexing with other conversation.

5.2 Accounting Events

Another function of the controller could be the maintenance of accounting information. Accounting may be done at various grains of detail. For instance, the controller might count elapsed time of calls and distance, as is often done with switched networks. However, finer events might better be used, such as the total number of packets delivered in the conversation, or the total number of bits. Such measures could easily be kept, and would more accurately represent the load that the conversation puts on the system. These would also be fairer measures, since they are better measures of the amount of conversation than is simple connect time, and they would not put a great burden on the user who is holding a connection but not using it heavily. Fairer still might be a combination of the two schemes.

The External Controller may select a set of events which the Voice Funnel will report to it so that the controller may

implement any accounting strategy it may choose from a broad spectrum. Thus, the accounting policy may be kept in the External Controller, isolated from the Voice Funnel itself.

5.3 Statistical Measuring

In addition to the purely accounting usage, the Voice Funnel can provide data about the current performance of the Voice Funnel application and the Butterfly Multiprocessor. This requires two kinds of information: data stream flow and processor/device utilization.

The utilization statistics will consist of periodic reporting of processor, device, and task statistics. These reports will provide the idle time statistics of processors, devices, and processes, and the queue lengths associated with them. From this information, the controller may produce reports identifying bottlenecks in the hardware and/or software.

5.4 Monitoring Traffic

The External Controller is in a position to monitor the traffic through the Voice Funnel. The data stream statistics will report the progress of packets through the Butterfly Multiprocessor. Each report item will include a datum identifier, time stamp, and event code. This information provides the controller with the data to measure how rapidly

packets are getting through the Butterfly Multiprocessor, and to determine when the Voice Funnel is throwing away packets. Note that the detail provided by this information may also be used as an audit trail and for debugging purposes.

6. Internal Controller and Support

The Voice Funnel will need internal modules in order to support the activity of the External Controller and to be able to continue to operate in its absence. An internal controller support module will be provided to offer a clean protocol level interface to a controller, and an internal controller will be provided. The internal controller may be either a very simple module to keep the system running when the External Controller is unavailable, or may be a functionally complete module so that no External Controller is needed. These modules are described in the following sections.

6.1 Controller Support Module

The controller support module is an internal host which communicates with the external or internal controller using standard network protocols, and interfaces to the internals of the Voice Funnel. The support module carries out the policy of the controller, and gathers and transmits the data which the controller requires. This enables the two controllers to be interfaced cleanly and simply as hosts, since all the difficult interface issues are centralized in the support module, which does not change for the two versions of the controller. In addition to providing a clean interface for the controllers, it provides a switch which enables the Voice Funnel to be easily

shifted from one controller to the other as needed, and, by communicating with the controllers using a network protocol, enables the External Controller to be physically removed from the Voice Funnel. Should it prove desirable to have a variety of External Controllers, the controller support module provides an interface that would make the exchanging and interfacing of the controllers simple.

6.2 Internal Controller

The internal controller would be a host implemented internally to lessen the reliance of the Voice Funnel on the availability of the external controller. It may be that the External Controller will not be available as soon as the Voice Funnel, or that it may be implemented at a remote site and suffer from failures over its communication link. The presence of an alternative controller inside the Voice Funnel will enhance the reliability of the system.

While a minimal internal controller is required for the Voice Funnel, a more extensive internal controller may be developed in the Voice Funnel software and interface to the network switch as a host. It would communicate over the switch to the controller support module using the same protocols as the external controller would. The controller support module would not be aware of which controller it was communicating with, except perhaps for its network address.

7. Optional Hosts

In addition to the required Voice Funnel network hosts, there may be several additional kinds of hosts, both internal and external, added as desired or needed. Of the hosts described below, we anticipate that we will implement only rudimentary versions initially, but that more complete implementation of these and other hosts may be desired later.

This strategy is supported by the protection and process structure of the operating system and by the use of the C language on the Butterfly Machine. Processes and protection in the operating system permit internal hosts to be written without untoward interaction or interference with other parts of the Voice Funnel. The use of C as the implementation language means that we may take advantage of available programs already written in C (such as a program which processes INTERNET TCP), and to permit hosts to be written without specifying whether the code will reside on internal or external hosts; the decision could even be a function of system load.

7.1 Conference Protocol Host

The conference protocol host, if provided, would control conferencing for a set of encoder hosts. In the NVCP protocol, a conference chairman is specified whose purpose is to provide centralized control for the conference functions (assigning

speakers, inviting, accepting, and removing conference participants, etc.). This function could be supplied by any host, including one of the participants of the conference. A useful alternative would be to provide a conference protocol host for the Voice Funnel so that the capabilities of a conference chairman need not be built into the encoders themselves.

The conference protocol host could be implemented as either an external or internal host. The present NVCP protocol is sufficiently simple that an internal host would probably be a more convenient implementation. However, as the protocol is expanded or replaced, it might provide the conference chairman with a more complex set of tasks, and we might then determine that an external host is more suitable, since most of its functions are outside the main line of processing for the Voice Funnel.

7.2 Simple Encoder Interface Hosts

In the design of the Voice Funnel we assume speech terminals which are quite intelligent devices. While such speech terminals are envisioned in the near future, currently voice encoders are used which are able to provide only a data stream and perhaps a few control signals. Currently available devices could not be attached directly to the network switch of the Voice Funnel.

The Voice Funnel can accommodate such devices only by providing a simple, low-level interface. Internal hosts would be constructed which would provide the specialized hardware interfaces required by the simple speech terminals, and would present the same logical interface to the network switch as would be provided by a speech terminal host.

These internal hosts would necessarily include a great deal of knowledge about the mechanics of speech transmission and voice protocols. Such a host module would have to incorporate reconstitution algorithms, which normally are resident in the intelligent speech terminal, and would require speech protocols to be built in, since they would be providing some of the control circuitry for the encoder.

7.3 Non-Speech Hosts

Since the interface to the Voice Funnel takes the form of INTERNET packets, it is reasonable to expect that external devices will be attached to the Voice Funnel which are unrelated to speech processing. The Voice Funnel provides a network interface and a capability for handling stream data very efficiently. We expect these facilities to attract others to use the Voice Funnel as a convenient entry to the INTERNET environment.

Hosts which are themselves interested in stream transmission would find the Voice Funnel a particularly attractive entrance to the INTERNET environment. Outside the speech community, we would envision stream service to be attractive for facsimile transmission, video signal transmission, and other high bandwidth applications.

For some of these applications, the available devices may not have the capability of acting in the role of network host. In this case, an internal host could be provided in the Voice Funnel with a specialized interface to the device. This would, of course, require more effort to attach the device, but might be an attractive alternative to providing an external host.

8. Protocols

The Voice Funnel will employ two different classes of communication protocols. The first will be used to communicate speech-related information from a single conversation. This protocol involves no information multiplexing, and is not concerned with any of the aspects of the communication medium over which it travels. The Network Voice Protocol and Network Voice Conference Protocol are of this class. The second class of protocol is concerned with the transmission and multiplexing of information, and is largely unaware of the contents or characteristics of the information being transmitted. The PSATNET protocol and the INTERNET protocol are of this second type. In our discussion we will refer to the first type of protocol as a high level protocol and to the second as a low level protocol.

8.1 Network Voice Protocol (NVP)

The Network Voice Protocol [COHE 76a] consists of two parts -- a protocol for transmission of speech parcels, and a protocol for transmission of control information regarding speech encoding characteristics and call connection. For each conversation, two logical channels are required in each direction, one for speech and the second for control.

Messages on the control channel comprise from one to three words. The first word identifies the message type, and any remaining words are parameters of the control message. Control messages are available for establishing connections, negotiating encoding parameters, and determining status.

Messages on the data channel are in a fixed format (see Figure 8.1-1). A message consists of a 32 bit header followed by an integral number of fixed length parcels (the parcel length is dependent on the encoding parameters used in the conversation).

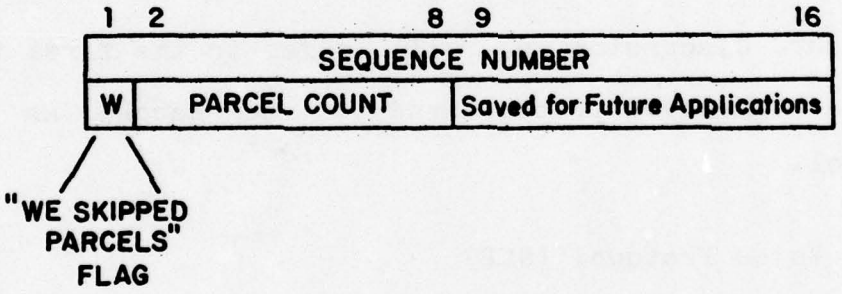


Figure 8.1-1 Network Voice Protocol Data Header Format

8.2 Network Voice Conference Protocol (NVCP)

The Network Voice Conference Protocol is an extension of the Network Voice Protocol [COHE 76b]. The data transmission part of the protocol is not changed in any way. The control part of the protocol, however, is extended to provide facilities for setting up conferences over the network. The NVP is a proper subset of the NVCP, which means that all the control messages of the NVP are available in the NVCP. When the NVCP is used to support a conference, it structures conversations around half duplex speech communication, rather than the full duplex channel in a simple two participant conversation.

In addition to the messages of the NVP, the NVCP includes several additional types of control messages. In a half duplex communication environment, permission to speak must be controlled by the protocols. A portion of the NVCP is assigned this task. In order to manage this, the concept of a conference chairman (one per conference) is introduced. The chairman may simply be a program which manages the conference control. The chairman's task includes setting up the conference, inviting and/or permitting users to join the conference, negotiating with users entering the conference concerning their encoding capabilities, and controlling rights to speak. The protocol is used by the chairman to communicate with participants (and potential participants) to carry out these functions. Included in the

control of speaking rights is the capability to limit the length of a speech, and to stop a speaker from talking in the middle of a speech.

8.3 PSATNET Protocol

The Butterfly Multiprocessor accesses the satellite through the PSAT computer, and it communicates to the PSAT using the Host/PSATNET protocol [BIND 78a, BIND 78b]. The PSATNET protocol provides two classes of service: datagram and stream. Datagram service is intended for general packet traffic. Each datagram is a distinct entity to the PSAT and contains all the status and routing information associated with the transmission. Stream access is intended for an environment in which a series of messages of similar characteristics will be sent periodically. Both stream access and datagram service permit broadcast of a message to a collection of receivers.

The stream access protocol will be our main use of the satellite link. This protocol provides requests to create, delete, join, and leave a stream, and to alter the characteristics of a stream. Within streams, the protocols provide for a stream member to send to all members of the stream or to any single destination. The stream provided is half duplex, meaning that only one host may safely transmit at a time; however, the protocol provides no explicit method of transferring

control of the stream. If a stream is to be shared, the members of the stream must develop their own protocol for stream reservation.

The datagram facility will be used for control (non-speech) traffic among the Voice Funnels. Unlike the stream facility, datagrams require no prior channel reservation, and may therefore be used by the Voice Funnel for coordination in setting up the streams. The datagram facility allows the user to specify priority, acceptable delivery delay, maximum holding time in the PSATNET, and reliability requirements. Unlike stream traffic, delivery of datagram traffic is acknowledged to the sender; thus datagram service is more reliable. However, datagrams incur more delay than stream traffic, requiring a minimum of two satellite traversals (about 0.5 second). Because they involve greater delay, datagrams are unsuitable for speech traffic. However, the delay is tolerable for control messages and the enhanced reliability makes them very suitable for that purpose.

Currently, both stream access protocols and datagram protocols involve six-word headers (96 bits). The current protocols are of a preliminary nature, and formats and services are subject to change.

8.4 INTERNET Protocol

The INTERNET protocol provides a technique for addressing messages across a variety of networks without being aware of the protocols of the networks which may be involved in the transmission of the message [POST 79a]. The INTERNET protocol uses a variable length header with a minimum length of ten words (160 bits). The INTERNET header is supplied by a network host which sends a message. The local network routes the message, including the the INTERNET header, to a gateway which interprets the INTERNET header and directs the message either to its destination or to another gateway (depending on whether it is addressed to a host on a network to which this gateway is attached). Before forwarding the message, the gateway replaces the previous local network header with a new header for the next local network to be traversed. The subsequent routing is at the discretion of the gateway. This procedure is repeated until the message, including the INTERNET header, is delivered to its destination host.

The INTERNET header provides for specifying several classes of service, source and destination addressing, header (but not message) checksum, message lifetime limitation, options to aid in the fragmentation and reassembly of long messages, and fields for other options. The source and destination addresses each consist of an eight-bit network identifier and 24-bit local address.

Every host has a 32-bit INTERNET address, and thus the INTERNET header addresses the source and destination hosts uniquely.

The INTERNET protocol will give the Voice Funnel the ability to access encoders across a wide variety of networks, through any network to which the Voice Funnel is attached. Initially, the Butterfly Multiprocessor will be attached to the PSATNET; however, we are anticipating the eventual need to attach the Butterfly Multiprocessor to other networks in such a way that it will have the capability of acting as a gateway for stream traffic.

8.5 Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) is a host-to-host protocol which may be used for reliable transmission between hosts in a network or internet environment. TCP is a higher level protocol than INTERNET, and TCP messages travel within INTERNET messages. TCP provides three basic services: increased reliability, flow control, and expanded addressing.

TCP assumes an underlying lower-level protocol, normally INTERNET, which provides unreliable service between hosts. TCP adds port addressing to provide a large number of addresses on each host. Additionally, using sequence numbers and retransmission, it provides a reliable transmission service with messages delivered in the order sent. Flow control is provided in order to prevent the receiver of messages being sent more than he can accept. TCP employs a variable length header with a minimum header length of 160 bits.

We do not expect to use TCP to support flow of speech data. Reliability is not needed, since retransmission of speech data over the PSATNET would violate latency constraints. Large address spaces inside the hosts are not expected to be needed for speech terminals. However, we will need reliable transmission for many control messages, and TCP may provide a simple mechanism for these transmissions.

8.6 Protocol Development

The protocols described above are in various states of flux. The NVP and NVCP protocols are expected to be replaced soon by NVP-II. NVP-II is not yet fully designed and it may or may not be available in time for use on the Voice Funnel. The PSATNET protocol is currently being defined and is expected to be available in time for the Voice Funnel. Changes in the PSATNET protocols will affect the Voice Funnel project, but are not expected to be great enough to alter our design in any significant way.

The remaining two protocols, INTERNET and TCP, are relatively stable. Some extensions to these protocols occur from time to time, but the basic structure of the protocols will not be altered. Our use of these protocols is less certain. TCP is not an ideal protocol for speech applications and may not be needed. Reservation facilities may be added to the INTERNET environment with an alternate protocol to the current INTERNET protocol; this protocol, called ST, is being developed by a group at Lincoln Laboratory, which hopes to incorporate it as a part of INTERNET protocol. It is expected that the combination of ST and NVP-II will obviate the need for INTERNET and TCP in the packet speech environment. The structure of the protocols managed by the Voice Funnel will depend on the pace of these developments and these ultimate results.

9. Difficult Issues

In the following sections we will consider a few of the issues involved in the design of an efficient Voice Funnel. Some of these issues depend upon design decisions for which we lack either the data or experience to resolve completely. We expect to keep our design flexible in order to be able to alter those parts of the design which prove wrong.

9.1 Message Format

The choice of message formats depends on several issues: (1) the desire to interface easily to other components employing standard protocols, (2) the need to address large numbers of speech terminals attached to a given Voice Funnel, (3) the desire to minimize message header overhead, and (4) the requirement for reliable operation. Consideration of the first two issues motivated the network structure which we have adopted for the Voice Funnel. The latter two issues have had considerable impact on our plans for the management of the satellite channel.

9.1.1 Encoder Addressing

The desire to attach a large number of speech terminals to each Voice Funnel requires that the speech terminals be either addressable devices on a host or hosts on a network. Since the speech terminals themselves are to generate INTERNET headers on

their speech packets, and since INTERNET headers address hosts on a network, the selection of the network structure seems appropriate. However, one addressing problem still remains: The combination of INTERNET header followed by NVCP header in the messages generated by the speech terminals leaves an addressing incompatibility. INTERNET headers address down to the host level, and each speech terminal is a host. The NVCP protocol, however, possesses no addressing capability of its own but assumes that the NVCP packets are addressed to separate ports for control and data traffic. Thus, there is no way for the NVCP protocol handler to distinguish control messages from data messages.

There are several possible solutions to the addressing problem. One obvious solution is to give each encoder two separate host addresses, one for data and one for control information. This would certainly solve the problem, but seems inconsistent with our notions of what it means to be a physical (or logical) host.

A second solution would be to insert another layer of protocol between the INTERNET and NVCP layers, perhaps TCP. This layer would provide a port number which would provide the needed multiple channels. This solution would also work, but requires the speech terminals to communicate using three protocols rather than two, and involves additional overhead in terms both of processing and number of bits transmitted.

The most desirable solution to the problem is the addition of the required addressing to the NVCP protocol. NVCP is expected shortly to be replaced by NVP-2. We strongly recommend that the addressing problem be resolved by the inclusion of a port address field in the NVP-2 protocol.

9.1.2 Stream Allocation

PSATNET streams could be allocated at three levels: each speech terminal could request enough allocation for its conversations; each Voice Funnel could request enough allocation for the set of conversations which it is transmitting; or each PSAT could allocate sufficient bandwidth for all its outgoing traffic. The closer to the PSAT that the allocation is performed, the more we can take advantage of statistical multiplexing. The idea of statistical multiplexing is that terminals will not transmit at all times, so that the total required bandwidth for a group of terminals will be much less than the sum of the required bandwidths of the individual terminals.

The PSATNET protocols require that stream bandwidth be reserved by the PSATNET host (i.e. Voice Funnel), rather than performing the function within the PSAT itself. The Voice Funnel can therefore either perform the allocation at the Voice Funnel or pass the task on to the speech terminals. We plan to perform

stream reservations within the Voice Funnel in order to take advantage of statistical multiplexing.

PSATNET streams pose two problems for the Voice Funnel: first, the Voice Funnel must establish how much stream capacity is required; second, the stream capacity must be negotiated between the Voice Funnel and the PSAT. In particular, the Voice Funnel must be able to handle the case in which it is refused the requested capacity or the PSAT unilaterally reduces the allocation of the stream to the Voice Funnel.

The problem of stream allocation policy is exacerbated by the limited data available to the Voice Funnel. The current protocols do not permit the Voice Funnel to receive information from the encoders about their perceived needs, nor can the Voice Funnel influence the encoding rates of variable rate encoders. This information and its negotiation are contained in the NVCP protocol layer, which is transparent data to the Voice Funnel. If the controller were used for call setup, the needed information could pass between the controller and the Voice Funnel.

Even with this information, the options of the Voice Funnel for channel allocation policy are quite limited. The Voice Funnel can operate on fixed channel allocations either large enough to handle the expected worst case (inefficient) or the

average load (which may lead to unreliable service). Of course, if its allocations were reduced by the PSAT below these levels it would be unable to respond effectively. Alternatively, the Voice Funnel could employ heuristics to predict loads and negotiate channel bandwidth according to these heuristics. The heuristics could be as simple as averaging the load over the last period of time, or complicated enough to consider time of day or source-destination pairs in the traffic, trying to recognize new connections as they are made, and disconnections as they occur. We do not feel that such techniques would necessarily track the traffic patterns quickly enough to succeed.

Another approach is to require the Voice Funnel to acquire more information about the activity of the speech connections. One way of doing this is to make the Voice Funnel a party to the NVCP control negotiations. The Voice Funnel could merely observe the NVCP traffic passively, alter or insert NVCP control messages to help determine encoding rates, or redefine the protocol to be a three party exchange. Such techniques tend to violate the spirit of protocol levels. A better, cleaner solution would be to include a new host-to-network protocol to allow specification of expected stream requirements. The Voice Funnel could combine such information from speech terminals and other hosts which generate stream traffic in order to better estimate its stream requirements. In addition, the protocol could include

network-to-host requests that the host reduce its traffic needs. It would then be the responsibility of the hosts to negotiate reduced channel requirements among themselves. The ST proposals are expected to include stream allocation of this sort, but not to provide the feedback from the network to the host level. We are hoping that the ST protocols will supply the basis for a solution to these problems, although the protocol will have to address difficult issues concerning the relationship between hosts and networks in providing more sophisticated flow control services than networks have heretofore provided.

9.2 Message Multiplexing

Since communication channels are an expensive resource, we would like to multiplex messages as much as possible in order to maximize our use of channel bandwidth. We have already discussed the multiplexing of the multiple INTERNET messages in one PSATNET message in our discussion of the PSATNET gateway. We would like to consider further multiplexing of messages since, for low rate speech encoders, the headers on a message could be far longer than the speech parcel itself.

The difficulty in performing much multiplexing in the Voice Funnel results from the interface level between the encoders and the Voice Funnel. Since the packets are formed outside the Voice Funnel, the Voice Funnel does not get the opportunity to form

INTERNET packets with speech parcels from a group of speech terminals. Moreover, since each speech terminal is a host, an INTERNET packet with parcels for a group of speech terminals cannot easily be formed since the Voice Funnel should deliver INTERNET packets identical to the ones it receives. Since each INTERNET packet may be sent with different parameters in the header, it would be difficult to merge a set of INTERNET messages into one large INTERNET packet with a single INTERNET header and then reproduce the original INTERNET headers to deliver to the encoders. Nevertheless, we intend to investigate what multiplexing and message compression may be feasible.

10. Summary

The Voice Funnel is a network designed to support packet-switched voice communication. It is designed to be implemented on a multiprocessor such as the Butterfly Multiprocessor.

The structure of the Voice Funnel is that of a packet-switched network. This facilitates the use of existing and developing protocols for the attachment of speech terminals as network hosts, and permits the extension of Voice Funnel facilities by the addition of hosts, both conventional hardware hosts and hosts developed as software modules in the Voice Funnel itself.

Connection of the Voice Funnel to the INTERNET environment will be through a gateway, a host implemented in software on the Voice Funnel. Initially, this gateway will connect to the PSATNET through two hardware interfaces to a PSAT computer. Many of the difficult problems we face are concerned with the management of the high traffic rates expected on this interface.

The Voice Funnel is expected to be a flexible and extensible system since its network structure permits extensions of the system by the addition of internal and external hosts. System controllers, an INTERNET gateway, hosts for interfacing simple encoders, and other computers are expected to be hosts of the Voice Funnel.

11. References

[BBN 78] "Specification for the Interconnection of a Host and an IMP", Bolt Beranek and Newman Inc., Report 1822, May 1978 (Revised).

[BIND 78a] R. Binder, "SATNET Host Access Protocol, Version 1", Bolt Beranek and Newman Inc., Research Note PSPWN 100, January 1978.

[BIND 78b] R. Binder, "Host/SATNET Stream Access Protocol, Version 1", Bolt Beranek and Newman Inc., Research Note PSPWN 104, April 1978.

[COHE 76a] D. Cohen, "Specifications for the Network Voice Protocol", Information Sciences Institute, Report RR-75-39, March 1976.

[COHE 76b] D. Cohen, "The Network Conference Protocol", Information Sciences Institute, February 1976.

[COLE 78] R. Cole and D. Cohen, "Issues in Packet Voice Interfacing", Information Sciences Institute, NSC Note 123, May 1978.

[LINC 77] "Network Speech, System Implications of Packetized Speech", MIT Lincoln Laboratory, Annual Report, September 1977.

[LINC 78] "Information Processing Techniques Program, Volume II: Wideband Integrated Voice/Data Technology", MIT Lincoln Laboratory, Semiannual Technical Summary, March 1978.

[POST 79a] J. Postel, "Internet Datagram Protocol, Version 4", Information Sciences Institute, Report IEN 80, February 1979.

[POST 79b] J. Postel, "Transmission Control Protocol, Version 4", Information Sciences Institute, Report IEN 81, February 1979

[SCHO 78] J. F. Schoch, "Inter-Network Naming, Addressing, and Routing", Xerox Palo Alto Research Center, Report IEN 19, January 1978.

Chapter III: The Butterfly Switch

Contents

Chapter III: The Butterfly Switch

Chapter III: The Butterfly Switch

1. Introduction

This chapter describes the design of the switch which will be used in the Butterfly Multiprocessor to interconnect processor nodes and to give each processor access to a common global memory address space. Many other groups have studied switches and multiprocessors. The result has been many forms of multiprocessors and many types of switches. The following three considerations will help to characterize our particular environment.

First, the Butterfly Switch is purely communications -- there are no processing elements in the switch. This differs from the use of networks of processors for both computation and communications.

Second, messages in the Butterfly Switch are independent. There is no effort made to coordinate the messages in the switch; rather, messages are sent whenever a processor happens to make a remote memory reference. Furthermore, there is no effort made to coordinate the actions of the processors at the memory reference level. This implies that there is no effort made to match the

configuration of the switch to the algorithm being processed on the machine.

Third, the purpose of the switch is to interconnect full and sovereign processors to a global address space. That is, the connected entities are processors, not pieces of an arithmetic unit.

The result of these considerations has been a switch with the following important characteristics:

1. The Switch is self-routing - each processing node has one switch port through which it sends all transactions, regardless of destination, and another port through which only transactions intended for this node arrive.

2. The Switch is in a basic sense serial - the transaction routing decisions are serial, proceeding from one node to another. Although there may be parallelism in the data paths of the switch, this parallelism is explicitly for the purpose of improving the performance of the switch. Since the data paths are naturally serial, data streams are easily and efficiently supported.

3. The Switch is highly regular - each switch node is identical. This favors an LSI implementation.

We should note that these characteristics come from the techniques used in the implementation of the switch rather than from the particular connectivity supplied by the Butterfly network. Some other interconnection, such as a crossbar, could have been used. However, the choice of the Butterfly network is particularly good since it keeps the number of nodes down, and still has performance comparable to the crossbar. Furthermore,

the Butterfly network has shown surprising adaptability in supporting the design improvements, such as the use of higher "base" switch nodes, which will be discussed in this section.

The Butterfly Switch is a surprisingly general structure. Far from being simply one internal component of a large machine, it is a data communications structure in a general sense. It provides services: data connection, addressing, flow control (or speed matching), and arbitration; it also has important characteristics such as speed, geographic extent, and protocols. Aside from these services and attributes, the switch is transparent to the rest of the machine. As a result, although we will use specific examples in this section, such as the connection of many processors to many memories via the Butterfly Switch, in fact the same structure could be applied to the interconnection of many processors in a loosely coupled multiprocessor. In the Butterfly Multiprocessor, the switch will interconnect the processing nodes and the global memory space.

We have been studying the Butterfly Switch for several years and have previously presented its design in BBN Report No. 3501. The reader who has read that report will find that we have duplicated some of the introductory material and may wish to skip over the following four sections, up to Section 6. This material has been updated to reflect the current design, but with the exception of the Serially Connected Crossbar in Section 4, the

changes are minimal. Although some repetition is present in the subsequent sections, for the most part they represent new work.

In the remainder of this chapter we will describe the overall structure of the Butterfly Switch and compare it to other switch structures. This description, however, does not necessarily represent the optimum design, so we will examine many variations on that design which may improve its size, cost, or performance. The selected design is summarized in Section 6.13.

To assure that this design is reasonable, and to begin to understand its performance in detail, we have performed system simulations which show that the performance of the Butterfly Switch is comparable to a similar crossbar switch. These simulations are reported in Section 7.

Finally, an implementation of the switch node is suggested and discussed at the end of this chapter.

2. Description of the Butterfly Switch

Before examining the properties of this new switch in detail, let us look at a simple example to see how it works. Figure 2-1 depicts a Butterfly Switch connecting eight sources (the circles on the left) with eight destinations (the circles on the right). Each of the (12) black dots represents a switch node. The function of a node is to examine transactions which enter on either of its two left-hand inputs and route them to the appropriate output depending on the first bit of the transaction (0=upper, 1=lower). The first bit of the transaction is then discarded before passing the transaction on to the next element.

To see how this works, please look at the figure. Here we have a short (8-bit) transaction entering the switch at the entry port labelled 1. Since there are eight possible destinations, we need 3 bits to select one. These are the first three bits of the transaction, leaving 5 "data" bits.

In a), we see that at the first switch node encountered (labeled 2) the first bit is a 0. Accordingly, the transaction is routed to the upper output after stripping the first bit as shown in b). At node 3, the first bit is one and thus the lower path is chosen (see c)). At node 4, the first bit is a zero and we choose the upper path which leads to the exit port labelled 5 in d). All that is left of our transaction are the 5 data bits.

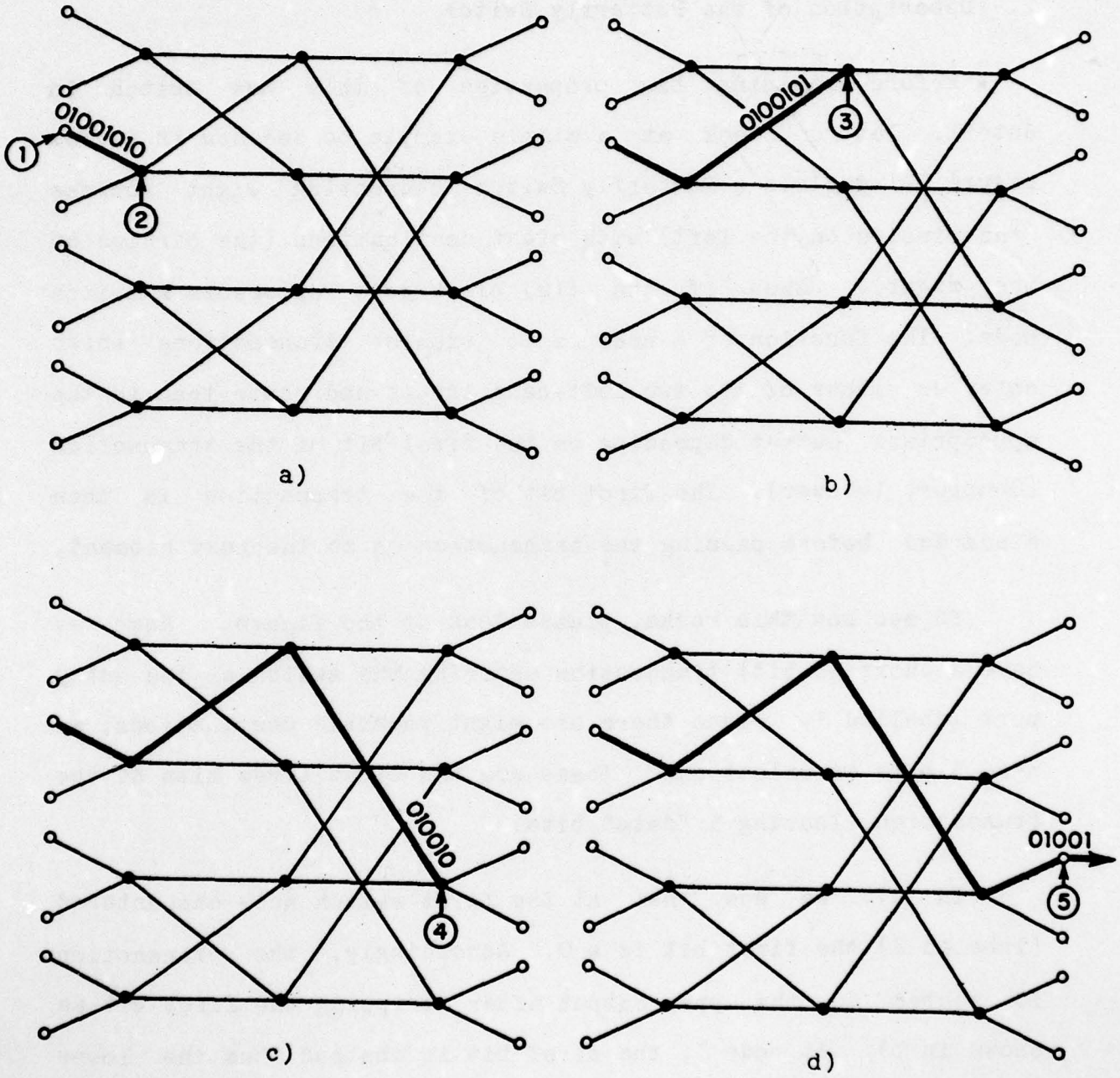


Figure 2-1 The Butterfly Switch

Let us now take the same transaction and have it enter the

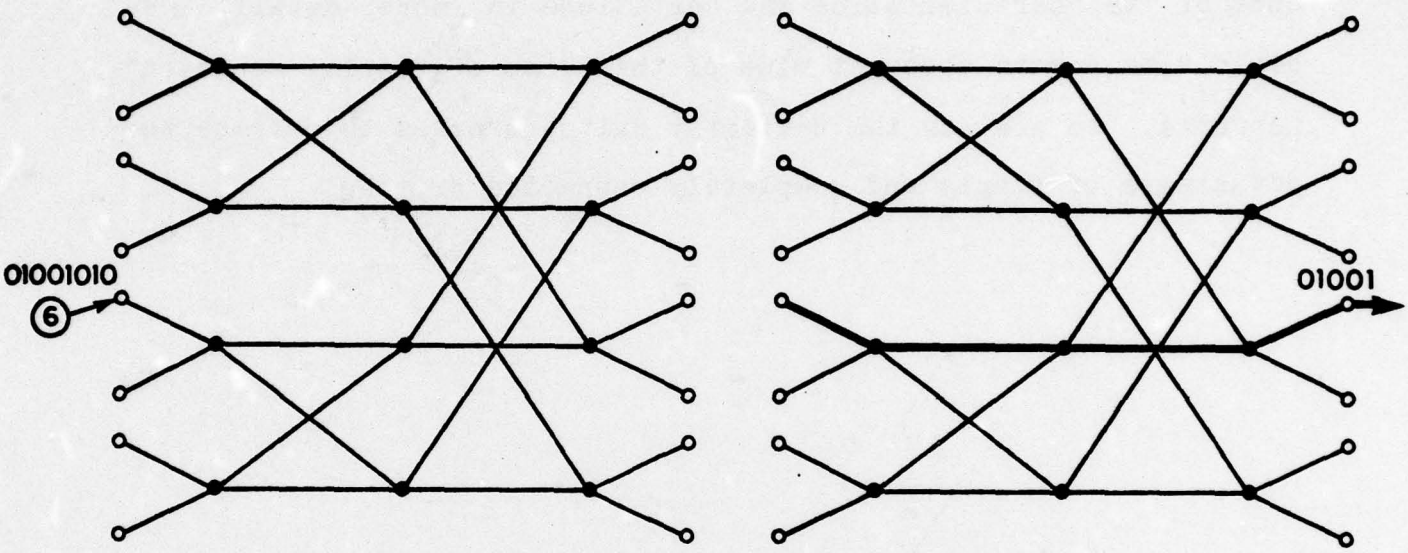


Figure 2-2 Another Source

network at another entry port, say number 6 in Figure 2-2. If we follow this out step by step as before, we find the pleasing result that it ends up at the same exit port (see b)). By checking other cases, it can be seen that there is a unique path for each entry to every exit port which is determined by the first three bits of a transaction, and furthermore that this implicit addressing of exits is consistent, independent of the entry port used.

With this brief example and demonstration of an important property of the Butterfly Switch in mind, we will now explore some of its characteristics and variations in more detail. We begin with a more abstract view of the class of "serial decision" networks, to see how the Butterfly Switch manages to combine the advantages of singly and completely connected switches.

3. Serial Decision Networks

In order to consider the value of the Butterfly technique, it is useful to examine the approach from a more abstract point of view. Consider the control (as opposed to data) part of the switch. It is made up of one or more "switching systems" that are responsible for getting a transaction from a source to a destination. For example, in a bus system the arbitration mechanism (who gets the bus next?), plus the various address recognizers (is this for me?), form the switching system. In a crossbar switch, there are several switching systems, typically one for each destination, all working in parallel. The key observation is that in each of these systems, the actual switching decision for each individual transaction takes place at only one instant of time. This implies that each decision must be an N-way decision, and thus that the switching system which implements the decision must have a fanout of N. There are two problems: N is large and N is not well-defined. It is very easy to make a switching node which decides between a small fixed number of alternatives. It is when this number gets to be very large and/or expandable that the design of the switching node and the resulting switching system becomes awkward, costly, or slow.

As an alternative, we could design the switch so that the switching decisions are not made all at once, but rather are spread out in time as the transaction wends its way towards its

destination. In the course of its journey, the transaction would encounter several switching nodes; at each one a decision would be made as to its route towards its destination. Each of the nodes could thus have a very small fanout (e.g., two) while still allowing the overall system fanout to be large and expandable. We call such a switch a serial decision (SD) network. An example SD network is shown in Figure 3-1.

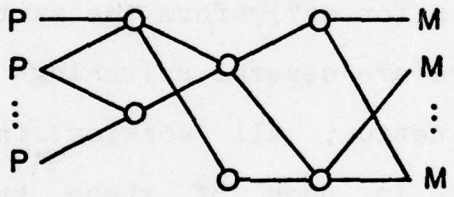


Figure 3-1 An SD Network

A good analogy can be made to a computer network like the ARPANET. Here the switching nodes are actually computers, and data paths are communications lines which span geographically significant distances. Transactions (called messages in the ARPANET) entering a system have the address of their destination contained within them. Each switching node (an IMP) routes each transaction along the best output path to reach its destination. Fanout from each individual switch node is small, but the entire network interconnects over 100 source-destinations.

The environment of a single computer, on the other hand, clearly involves a different set of concerns than does the computer network. A serial decision network with a simple, regular structure like the Butterfly is more easily implemented in the single machine case. Geographic constraints which force the connectivity of large networks to be less than ideal vanish in this application. Similarly there is no need for elaborate protocols or complicated schemes for routing and retransmission in light of the relative simplicity of the environment. Thus the switching nodes become very simple devices. A programmed CPU is not needed; the necessary algorithms are easily implemented in dedicated hardware. We do not need to buffer an entire message -- only a bit or two will suffice.

Another important difference between the geographically distributed and single machine case concerns the contents of the transactions. In a network we send "messages" between essentially independent computers. We could do a similar thing in the single machine case: think of the system as a collection of loosely coupled independent machines which can send messages to each other. On the other hand we have been painting a picture of a more tightly coupled machine in which the "messages" through the network are the actual requests by the processors to fetch and store the contents of memory locations. A transaction now consists of an address and some data to write into that location,

or a request to read a particular location, or the contents of a location being returned to the processor who requested a read.

Notice that we have made a critical change from the usual scheme in which the processor establishes a path to memory through whatever switching there may be and holds that path until the memory processes the request and the desired data is returned over that path. In the approach proposed here, each stage of the switch is busied only long enough to pass a request on to the next stage. Data returning from memory to processor is contained in an independent transaction which must find its own way through the network. We call this approach "transaction switching". (See the discussion of circuit switching vs. packet switching in [SWAN 75].)

Now that we have outlined the general idea of an SD network, we can turn to the question of the optimal implementation of such a network. In doing so, we specify the properties of the switching nodes and how these nodes would be connected together. Some of the important properties of a system constructed in this way are:

1. Expandability - How easily are new nodes added? Are there any limits to growth? Are there any significant cost jumps as specific sizes are reached?

2. Routing - What sort of intelligence has to be built into the switch nodes to route transactions towards their destinations? Can it be simple and fixed or does it need to be a more complicated dynamic algorithm?

3. Parameterization - It would be advantageous if all of the nodes in the network could understand all they needed to know about their connectivity and topology without any external parameterizations such as switches or jumpers.

4. Homogeneity - Similarly, it would be helpful if all switching nodes could be identical.

5. Balance - Does the system lend itself to being well-balanced? Are there any obvious bottlenecks? Can the system be tuned to fit its environment?

6. Reliability - Does the network design lend itself in an obvious way to reliable operation, even if some of the switching nodes and/or pathways are down?

7. Implementation - How well does the design submit itself to integration? Does it take advantage of any of the new technology? How well does it partition?

8. Interface - How easy is it to connect to the switch, in terms of both hardware and software?

The Butterfly Switch is one implementation which satisfactorily answers many of these questions. In the next sections, we will examine its properties in more detail.

4. Alternative Switch Structures

Consider the problem of connecting many processors to many memory units. The classical approaches to this kind of problem group themselves in two distinct categories called "completely connected" and "singly connected" switches.

Completely connected switches are characterized by every source having an independent path to every possible destination. An example is a classical crossbar switch.

The crossbar switch can also be distributed, as in the Bus Coupler switch used in the Pluribus multiprocessor [HEAR 73] (see

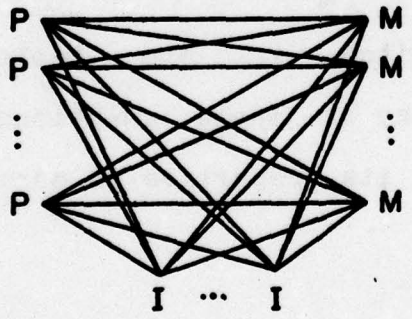


Figure 4-1 A Distributed Switch: Pluribus Bus Couplers

Figure 4-1). The completely connected approaches provide high potential bandwidth since each source has its own private path that it can use independent of any source. However, the number of switching nodes and therefore the cost of the switch goes up

as the product of the number of sources and destinations. Therefore, as systems get larger the cost of the switch tends to dominate the overall system cost.

A singly connected switch, on the other hand, has only a single path which connects all elements of the system and upon

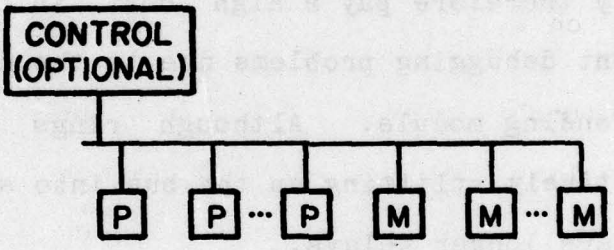


Figure 4-2 Bus Switch

which all communications flow (see Figure 4-2). Several sources share this path according to some discipline, e.g., time division multiplexing or asynchronous arbitration. Examples of this form are the bus [CHEN 74, NSC 77], the ring [FARB 72], the Ethernet [METC 75], and broadcast channels [ABRA 69]. Of course, there may be more than one such pathway for reliability or bandwidth reasons, but basically the number is small. This switch expands smoothly since its cost goes up linearly with the number of elements it connects, but it is severely bandwidth limited. When the number of elements exceeds the necessarily finite bandwidth of the pathway, this scheme is no longer usable. More busses may

be added, but this impacts each of the connected elements which then must have an arbitrary fanout.

The particular implementations of singly connected switches have their own disadvantages. Busses are hard to extend beyond certain limits, unless they are very carefully designed for this goal (and frequently therefore pay a high cost in efficiency). Busses also present debugging problems due to the difficulty of identifying the offending module. Although rings solve these problems by effectively splitting up the bus into short pieces, they consequently have longer delays.

Between completely connected and singly connected switches are what we call "serially connected" switches. As one example, imagine that a large, say 100 X 100, crossbar switch were divided into two smaller switches, one 100 X 10, and the other 10 X 100 as shown in Figure 4-3.

This modification has reduced the number of crosspoint elements required from 10,000 to 2,000 -- a large reduction. Unfortunately, this change has had the effect of introducing sharing of switch paths between sources at a point which is within the switch rather than only at the destination. As a result, the maximum potential bandwidth of the switch has been reduced by a factor of ten.

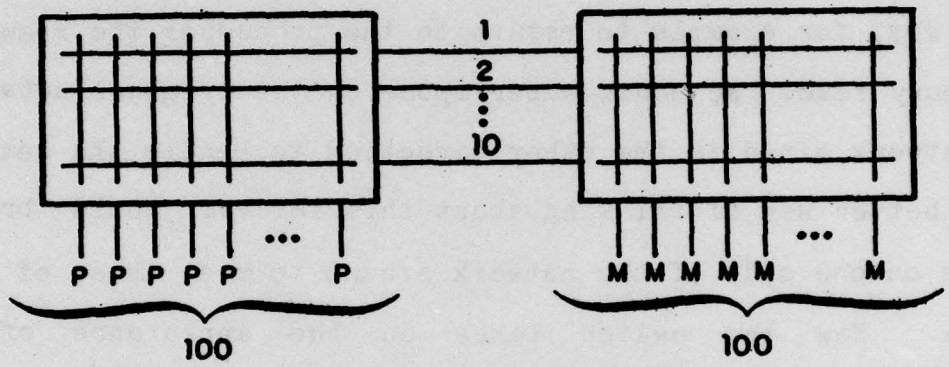


Figure 4-3 A "Serially Connected" Crossbar

Like the divided crossbar, the Butterfly Switch is serially connected. However, its interconnections eliminate the loss of maximum bandwidth as will be shown below.

5. The Switch Revisited

We have so far been talking only about transactions from processors to memories. Transactions must, however, also flow the other way, for example to return to the processor the results of a memory read. We could superimpose on the original network another network aimed in the other direction to handle the return flows. A better way of thinking about this is: we could bring the stubs on one side of the network around to meet those of the other side. Now the switch takes on the appearance of a

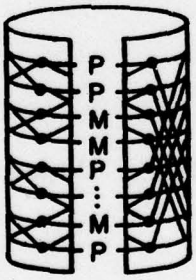


Figure 5-1 The Butterfly as a Cylinder

cylinder, as in Figure 5-1. The processors and memories are all located along a generatrix of the cylinder sending their outputs to the right and receiving their inputs from the left. All transactions flow in a counterclockwise direction as viewed from the top of the cylinder.

We now notice that the processors and memories, which up until this point have been pictured on opposite ends of the switch, are next to each other. We can then take the next step of combining a processor and some memory in a unit which we call a "processing node". These nodes interface to switch entry ports. Now the picture looks like Figure 5-2. The switch cylinder has become a very high bandwidth communications path between nodes.

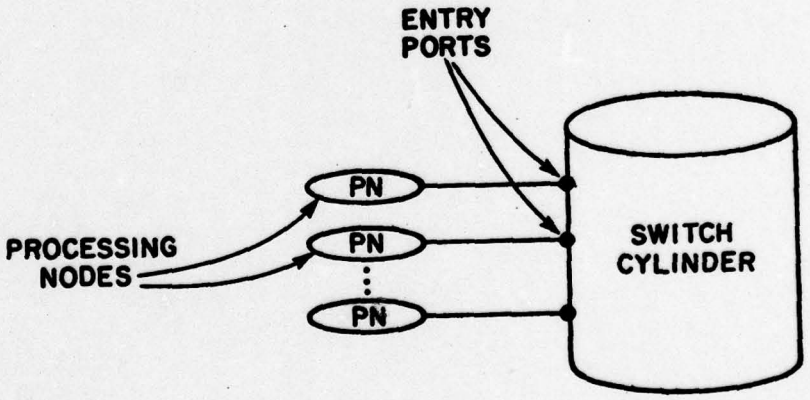
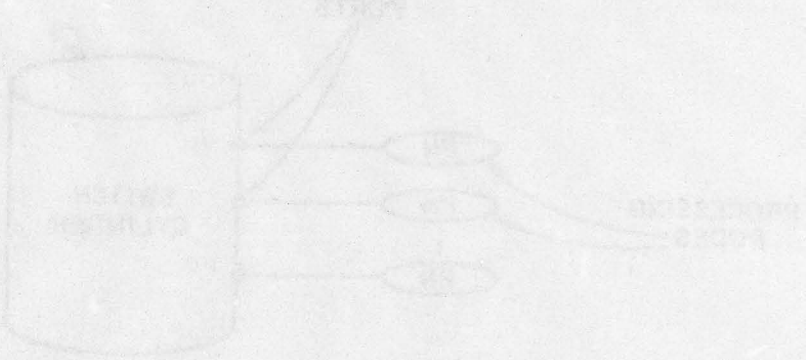


Figure 5-2 Processing Nodes

Notice that even though the memory is now segmented into a piece in each node, it is all still globally accessible and we can ignore the association between the processor and the memory

in the same node. On the other hand, now that a processor has some memory within the same node, there is a potential direct path to that memory without going through the switch. Now the memory can be used both as a "local" and as a "common" memory, and in fact can be dynamically allocated between these uses.



6. Important Design Variations

The preceding discussion presented the basic structure and operation of the Butterfly Switch. In implementation, many important variations remain. This section presents some of the more interesting variations which we have explored. After this presentation, we will summarize the switch as we would implement it now.

The design of a Butterfly Switch is complex enough that many of the topics interact. We therefore apologize in advance for discussing some topics before they have been properly introduced.

6.1 Conflict Resolution Strategies

Conflicts arise when two transactions at a particular switch node want to use the same exit path. Clearly both of them cannot, so an appropriate strategy must be chosen to resolve the conflict. An easy algorithm is to let the loser wait until the winner has completely passed. This was thought to be a workable though perhaps inefficient approach. We have found, however, that this simple algorithm cannot be used in a uni-directional switch since it leads to a deadlock (as we will see later). We have also found that this deadlock does not exist for a switch which has bidirectional data paths.

An alternative algorithm is to make the losing message retreat back to its source and retry some time later. This strategy does not necessarily lead to deadlocks (although one must design the switch interface with care as we will discuss later). Furthermore, this strategy seems to have better performance than the "wait" strategy because it reduces the "profile" of a message. That is, it reduces the number of secondary conflicts that result from the original conflict.

To solidify this a little, see the example in Figure 6.1-1, and assume that the simpler "wait" strategy is being used. Here Transaction A is trying to get to exit port B via the dotted line. It is blocked, however, at node C by Transaction D which is already using the upper output of C. Transaction A must therefore wait until Transaction D has passed through. Transaction E is trying to get to F via the dashed line. It runs into a conflict at G because the lower output is busied by Transaction A. So now Transaction E must also wait, first until Transaction D passes, so that Transaction A can start, and then until Transaction A goes by. A more global scheduling policy would have noticed that Transaction E could be running simultaneously with Transaction D, but this information is impossible to obtain in this distributed system.

If we use retreating, the blocked message will present a much smaller average profile to obstruct other messages,

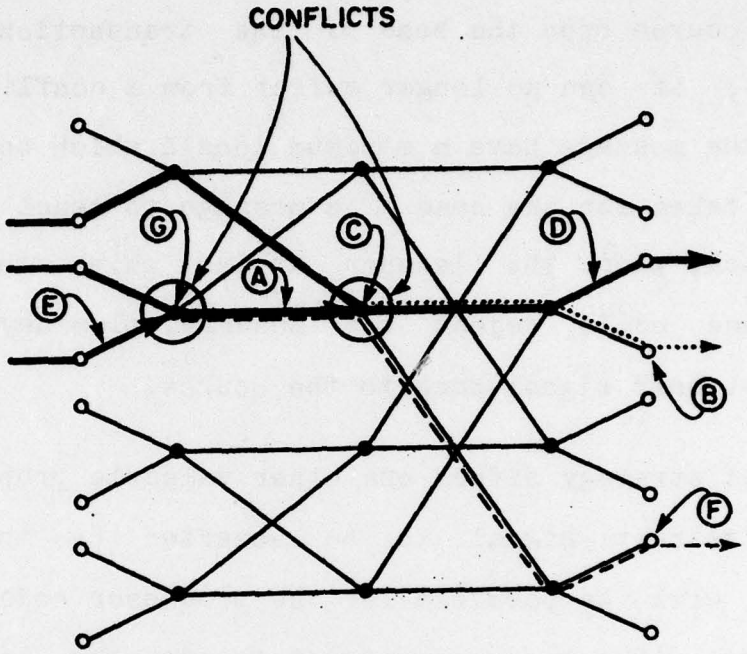


Figure 6.1-1 Secondary Blockage

increasing the overall bandwidth of the network. This in turn decreases the average delay of a transaction significantly, especially when the net is heavily loaded. This is important even if the network as a whole is lightly loaded, as there will be areas of high contention due to locality of memory requests.

In the "retreat" strategy, a transaction which encounters a conflict "retreats" to its entry port (is cancelled) and then retries. This is possible as long as the tail of the transaction

has not left the entry port, since otherwise the retreat path is not known. Of course once the head of the transaction reaches the exit port, it can no longer suffer from a conflict. This requires that the message have a minimum length which corresponds to the time it takes for the head of a message to reach the other end of the switch, plus the latency during which the remote switch interface could reject the message, plus any delay in passing the "retreat" signal back to the source.

The retreat strategy offers one other valuable property. If we allow the retreat signal to be asserted by the switch interface, it will be possible for the processor node to get a peek at a message without any commitment to actually accept it. It is this feature of the retreat strategy which ultimately allows us to design a switch interface without deadlocks.

An important system concern is the maximum delay to send a message through the switch. This determines our ability to establish timeouts for lost or broken messages. With the "wait" strategy, the worst case is that all sources simultaneously address the same destination (ignoring deadlocks!). Although a large number, this is a hard upper bound. In the "retreat" case, on the other hand, there is no upper limit. It is possible, though unlikely, to have a transaction take five minutes or a week to traverse the net. As a result, we may never know whether something is truly broken or whether we have just been

unlucky, thus leading to debugging problems. However, since any system of this type must deal with intermittent failures anyway, it may be possible simply to treat this as another source of noise in the system.

6.2 Parallel Data Paths

Clearly, the bandwidth of the Butterfly Switch is going to be an important parameter of a Butterfly Multiprocessor. The simplest way to improve this bandwidth is through parallelism in the data paths of the switch. This also has the advantage of amortizing the overhead of the control portion of the switch -- the control logic in an MSI implementation, and the control pins in an LSI implementation. At times, we will use the term thickness as a synonym for data parallelism when referring to the dimensions of a Butterfly Switch.

It is not necessary to go to extremes, when introducing parallelism, by designing switches which permit the transaction to be only one tick long; a small amount of parallelism (such as 2 or 8 bits parallel) produces a switch in which one path has a bandwidth of many tens of megabits. Large amounts of parallelism become unattractive when the marginal system advantage of an extra data path becomes less than the marginal increase in the cost of the switch.

Another problem can arise with parallel data paths. Wide parallel paths cannot be supported with only one LSI package per switch node because of pin limitations. This suggests a structure consisting of a control chip and one or more data chips. A more interesting structure is one which has several identical chips each making the same routing decision. However, consider the question of addressing errors, that is, failures which cause a switch element to make an incorrect routing decision. These are insidious errors; mistakes in the data are more easily caught and handled.

The following discussion is somewhat hypothetical in that the initial implementation is only 4 bits thick and is expected to fit in one module, and thus will not be prone to the errors described. The example is presented because it is interesting and shows a surprising form of flexibility available in selecting data path parallelism.

Example: imagine that the switch is 4 bits thick and each chip is two data bits wide. Thus each switch node consists of two chips. Rather than thinking of these as a master and slave (with respect to routing control), give the address to both chips and have them make the routing decision in parallel. This eliminates the necessity for a wire (and its attendant delay) between the master and slave to indicate the addressing decision. But now assume one of these chips is broken and makes a wrong

routing decision. We now have two message fragments going off in different directions. When these fragments reach their destination, the "end-to-end" checksums will fail and we will know that something is amiss, but it is worse than that. On their way through the network from the point of failure, the cleaved message will no doubt conflict with other messages. When a fragment of this cleaved message conflicts with another whole message, it splits that message as well. Now we have more cleaved messages in the system which continue to spread the disease. From the outside, all we can see is great confusion. There is no good way to tell which is the offending element nor even to stop the spread of the plague.

It would be very beneficial to detect at least some of these errors at the switch node level so that we have a chance of identifying the offender before the effects spread. One idea is to have the switch element chips do some error detection based on some redundancy in the transaction, but the problem remains as to what to do when such an error is detected. Lighting up a light (or its equivalent) seems a good thought, until we realize that very quickly almost everyone's light will be lit. We really need a way of telling whose was the first light lit. Better yet, we need a way of containing the error to the vicinity of the failure, so that the rest of the system can continue.

A proposed solution: imagine that each of the chips which make up a single switch element are connected by a set of bus wires, one for each of the inputs. When a switch chip detects the beginning of a message (the start bit), it pulls down the bus wire corresponding to that input. Chips always watch their bus wires and, if they ever notice that a bus is down and they are not themselves pulling it down, they a) refuse to listen to the corresponding input any more, b) cancel any action on behalf of that incoming message, and c) pull down on the bus forever (until explicitly reset).

This works on the assumption that if the chips of a switch element disagree on the beginning of a message, then there has been some kind of addressing failure in the preceding switch element. The effect of the bus rules is that the failed chip will be isolated from the network until reset. We now have a stable state where traffic through the bad element is blocked and all else is unaffected. It should now be easy to identify the culprit and configure around and/or replace it.

At the cost of (base) pins per chip, we have a scheme which has all the desired properties: routing errors are quickly detected and contained to the immediate neighbors. The error detection and correction processes take place in a "healthy" chip; we do not have to count on correct behavior of a chip which is failing. An intermittent failure is latched, so that it is

easy to detect and isolate. All chips are identical and unparameterized. It does not appear that the inter-chip communication will materially slow down the cycle time of the switch.

A nice reflection on this scheme is that we are using to advantage the originally perceived weakness of the design. That is, the parallelism of the switch was the thing which originally made us concerned about cleaved messages. This scheme, however, uses the very fact that independent decisions are being made, in a kind of voting process, to insure that the actions taken are correct. By requiring that all chips of a switch element see the beginning of a message at the same time, and assuming that independent chips are unlikely to make simultaneous mistakes, then we should detect all kinds of address routing errors.

6.3 Switch Node Base

Previous descriptions of the Butterfly Switch have assumed that the switch nodes have two inputs and two outputs. This choice is somewhat arbitrary; in fact, a switch node with any number of inputs and outputs (greater than 1) could be used. Indeed, we feel that a switch node with 4 inputs and 4 outputs is much better because it reduces the number of switch nodes by a factor of 4 and the number of interconnections by a factor of 2. Indeed, a switch of base 8 would reduce the number of nodes

further. However, as we will see, large bases lead to less modular switch sizes and larger switch node implementations. A

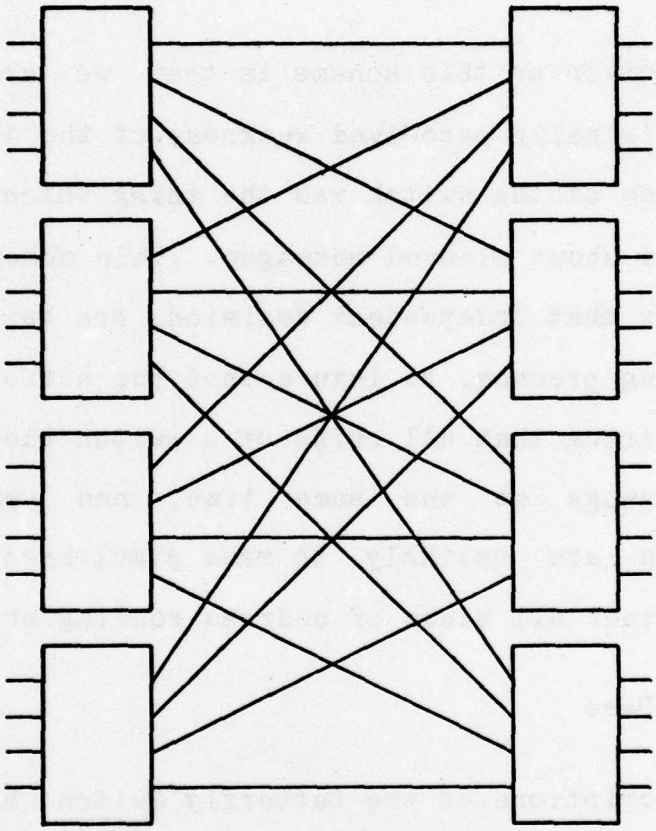


Figure 6.3-1 A Butterfly Switch of Base 4

choice of 4 for the base seems a good compromise.

We will call a switch node which has B inputs and B outputs a switch node of "Base-B" because the number of nodes in a Butterfly Switch with N ports is:

(N/B)*Log[Base B](N)

Unfortunately for bases other than two, the easily recognizable structure of the Fast Fourier Transform is lost. For example, Figure 6.3-1 shows the interconnection pattern for a 16 X 16 Butterfly Switch constructed from base-4 switch nodes. For comparison, Figure 6.3-2 shows a 16 X 16 Butterfly Switch using base-2 Switch Nodes.

From these two figures, it is obvious that the switch with the higher base has fewer nodes and fewer wires. We can quantify the impact of the selection of a base on the size and structure of the switch in hopes that this will lead us to an optimum base for the Butterfly Switches we will construct. These calculations assume that the number of ports is an integral power of the base of the switch. Other numbers of inputs and outputs will be discussed later. The calculations count the number of input port wires but not the number of output port wires. The difference is N wires.

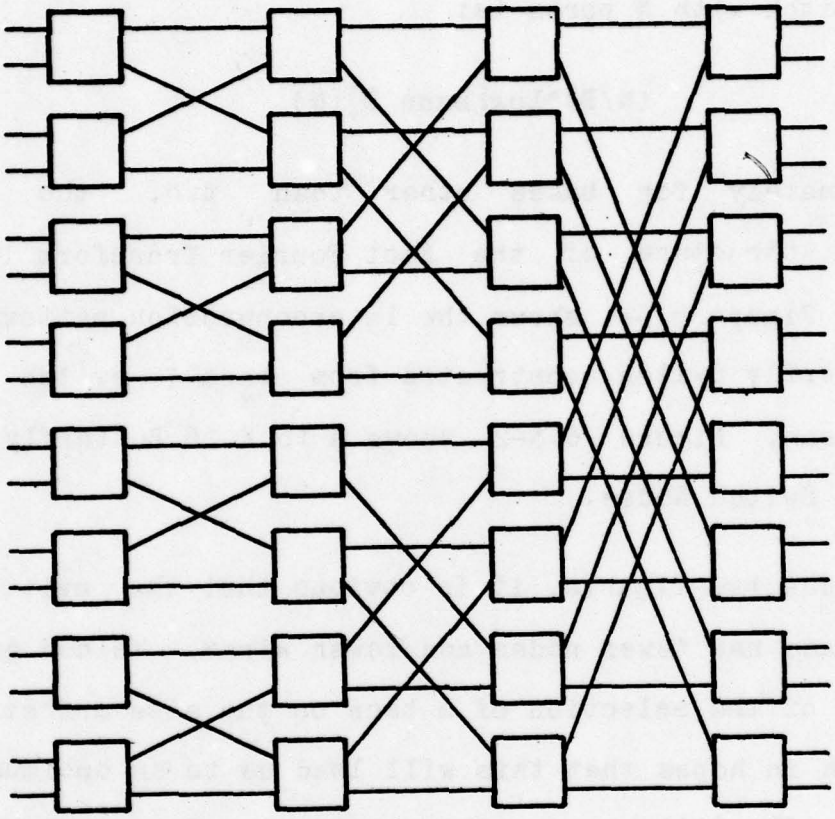


Figure 6.3-2 A Butterfly Switch of Base 2

Assume:

S = Number of switch nodes
N = Number of ports into the switch
B = Base of the switch
C = Number of columns
W = Number of interconnections

Then,

$$C = LOG[Base\ B](N)$$
$$S = (N/B) * LOG[Base\ B](N)$$
$$W = B * S$$

If we wish to compare switches of two bases, we can derive the ratios of the number of switch nodes and the number of wires in two switches as follows, assuming an equal number of input ports:

$$S1/S2 = (B2 * LOG(B2))/(B1 * LOG(B1))$$

$$W1/W2 = (LOG(B2)/LOG(B1))$$

The following table compares base-2 switches with higher base switches. In each case, the table entry gives the improvement provided by the higher base. For example, a base-2 switch has 12 times as many switch nodes as a base-8 switch.

BASE	S1/S2	W1/W2
2	1	1
4	4	2
8	12	3
16	32	4

Thus a switch with a larger base has fewer switch nodes and fewer interconnections. Switches with large bases have a further advantage over those with small bases in that a single switch node can be built as a B X B crosspoint switch. This reduces the number of conflicts in the network, thereby increasing the performance of the switch.

While the reduction in the number of collisions is an important advantage of larger bases to the performance of the network, we should also note that a higher base implies fewer columns. This will decrease the actual delay across the switch to a small extent.

While these arguments suggest that the largest possible base should be selected, there are problems with large bases. In particular, the Butterfly Switch does not grow as smoothly as we had earlier expected. Although it is practical to make a large range of switch sizes from the same switch node, there are rather sharp growth points where the size of the switch must be increased significantly to add one more port. We will discuss these issues more under the subject of partial switches.

In light of these considerations, we expect to build our switch using base-4 nodes since it achieves a good compromise between the complexity of the switch node and the improved performance of the switch.

6.4 Extra Columns

As presently described, the switch network is vulnerable to switch node failures. Intuitively, the failure of an individual switch node can be seen to affect a wedge of switch nodes and entry points, emanating out in both directions from the failed node. This is shown in Figure 6.4-1. If we exclude from our

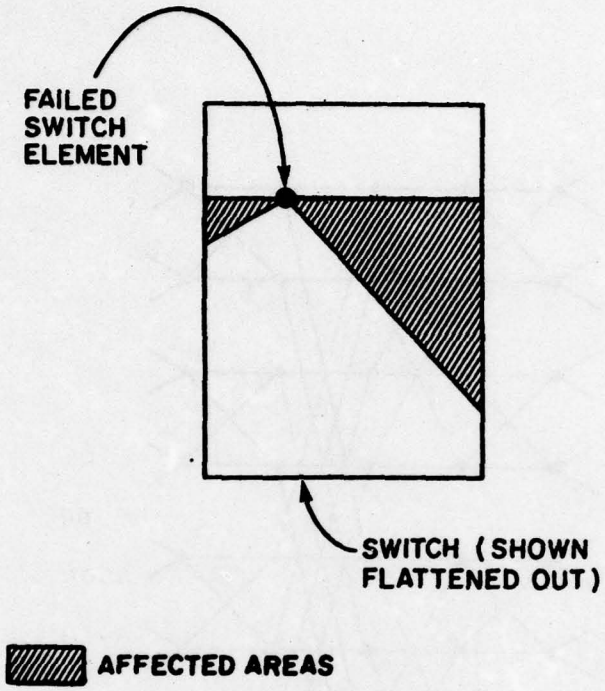


Figure 6.4-1 Effects of Switch Failure

system the sources and destinations connected via the entry points in either the left-hand wedge or the right-hand wedge, the rest of the system will still be fully connected. Thus we can see that the severity of a failure increases the closer the failed node is to the center of the switching network. A failure at the periphery of the switch might only take out one or two processors, not a very serious occurrence in a large system.

A good way to make the switch network more resilient against failures is to introduce some redundancy by adding some extra

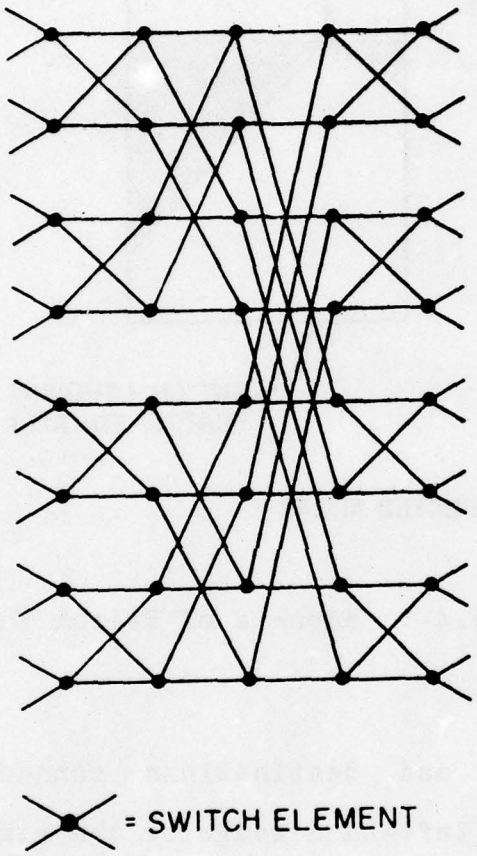


Figure 6.4-2 Extra Paths

columns of switching nodes as shown in Figure 6.4-2. This switch has multiple paths between each source-destination pair. Another way of saying this is that each destination has several

addresses, any of which can be used to reach that destination. In the example case depicted in the figure, the high-order bit of the address can be of either sense and still access the same destination. We now have a situation in which a failure of a node in the interior of the switch (that is, excluding nodes in the leftmost and rightmost columns) does not affect the connectivity of the switch. We have achieved this increase in reliability at the relatively low cost of adding only one column to the switch.

The correct structure (i.e. connectivity) for the extra column is to add to the right side of the switch a column which has the same connectivity as the column at the far left side of the switch. Such a configuration is shown in the figure. This algorithm works because the effect of a switch node is to permit a transaction to move from one row of switch nodes to another. In the Butterfly Switch, each column selects an orthogonal set of such moves. The extra column which has the same connectivity as another column furnishes an alternate method of providing such motion. Thus, the best way to protect against the largest number of switch node failures is to place these two columns as far apart as possible.

Indeed extra reliability can be achieved by adding more extra columns. However, a pair of extra columns only provides protection against failures in nodes which lie between them. So

a second extra column provides no protection against failures in the outer two columns on each edge.

We should note, in defense of a single extra column, that the switch node failures which are not covered by the extra column are those which only affect the small and well defined sets of processor nodes which are attached to each individual switch node.

How do we deal with these alternate paths to a given destination? They can be handled either at a hardware or a software level. The hardware in the switch interface, for example, could be designed such that if a transaction fails to reach its destination, an appropriate address bit is changed and the transaction retransmitted. At a software level, the alternate paths would manifest themselves as several addresses for any given destination. As a processor notices that it can no longer access a destination by one address, it switches over to an alternate. These software functions can also be supported by hardware if memory management hardware is used to perform the required transformation.

In all, the hardware approach is probably best since recovery is simple for hardware to implement but difficult for software as the indicated approaches would consume half of the limited physical address space in the current design.

These alternate paths might also be useful in decreasing the delay due to conflicts. If a conflict is encountered along one path, the hardware could try the alternates until a clear path is found. In some ways, this is just a shorter term implementation of the reliability retry described above.

6.5 Depopulating the Inputs

The load on a Butterfly Switch comes from the processors which are attached. One way of combating the decrease in performance of the Butterfly Switch as the load increases is to reduce the load on the switch by reducing the number of connected processors without decreasing the number of ports. We refer to this as "depopulating the inputs".

We have not examined this idea in great detail. However, except as a technique to be used in the next section, it seems that any extra parallelism introduced into the switch through depopulating the inputs would be better applied in some other way such as by introducing more parallelism into the data paths.

6.6 Partial Switches

As we have noticed before, Butterfly Switches have certain preferred numbers of ports which result in complete switches. By complete, we mean that in the switch, there are no nodes which have unused inputs or outputs. These numbers are a function of

the base. The number of ports in a complete switch is B**i where B is the base and i is an integer. The progression for various bases is as follows:

Base	Number of Ports (N)										
2	1	2	4	8	16	32	64	128	256	512	1024	...
4	1		4		16		64		256		1024	...
8	1			8			64			512		...

If N is not an integral power of the base, then the structure of the switch is that of a switch which is "the next size larger": that is, one with B**i ports where i is the next larger integer. This leaves a switch which has many unused ports, and potentially even unused switch nodes. There are two choices for dealing with these partial switches. Either we can remove the unused switch nodes from the switch network, or we can depopulate the switch inputs.

Figure 6.6-1 is an example of a 10 X 10 switch which has had the unneeded switch nodes removed. The pattern of unneeded switch nodes is much more complex in larger switches. Furthermore, the savings in switch nodes is often small. As a result, this approach makes sense only if switch nodes are expensive.

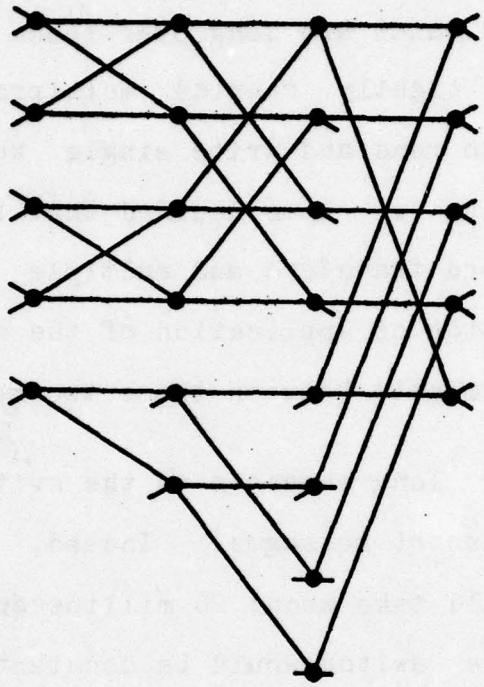


Figure 6.6-1 A Partial Switch

The alternative of depopulating the inputs does achieve some improvement in the performance of the system, and is simpler to implement from a system standpoint -- it is the one we would expect to apply to at least the initial systems.

6.7 Long/Short Messages

The pipelined structure of a Butterfly Switch implies a relatively large initial delay in setting up a transfer, followed

by a very high transfer rate. This means that the Butterfly Switch favors messages which are long over those which are short. The structure of a tightly coupled multiprocessor, however, requires the ability to read and write single words across the switch. As a result, we have decided that the hardware will support both single word transfers and multiple word transfers. This decision is a matter of application of the switch, since the switch need not discriminate between these two message types.

The presence of long messages in the switch may adversely affect latency of the short messages. Indeed, a transfer of, say, 64 kilowords, would take about 26 milliseconds, during which one path through the switch would be constantly occupied. We would expect this to block many short messages and stop many other processors.

The simplest solution to this is to divide long transfers into shorter pieces, of say 16 words. This has four further advantages:

1. The switch interface implementation is simplified because the messages can be assembled in hardware buffers. This also decouples the switch rate and the processor node rate.
2. With an appropriate switch interface design, the local processor can access remote memory locations during the transfer of a long data block.
3. Shorter messages improve throughput when switch transmission errors occur.

4. Should the processor need the full bandwidth of the memory for fast interrupt servicing, the sectioning of long data blocks into short data blocks may allow the processor to temporarily stop a data transfer during the interrupt service routine.

6.8 Bidirectional Switch Paths

The Butterfly Switch can be constructed such that the processor establishes a path to memory and then holds that path until the memory processes the request and the memory's reply is returned over that path. In this scheme, the data wires inter-connecting the switch nodes are first used to establish the processor-to-memory path and to send the remote memory access request message. After the memory processes the request, the data flow direction of the path between processor and memory is reversed and the memory sends the response message back to the processor. This is called the "bidirectional switch". The advantages of the bidirectional switch are:

1. While the request, or forward transaction, suffers from conflicts, the returning transaction does not.
2. It is not necessary to transmit the address of the sending processor node for the purpose of addressing the reply.
3. A reasonable set of values for remote memory access probability, switch clock, memory access time, etc., indicate a reduction of remote memory access time by as much as 40%.
4. The switch interface is somewhat less complicated.
5. There are no deadlocks possible even using the wait strategy for switch node contention.

The disadvantages of the bidirectional switch node are:

1. The profile of a transaction is potentially increased because paths through the switch node are held at times when no data is being transferred.

2. An extra control wire interconnecting switch nodes is required.

3. The switch node increases in complexity by about 40%.

4. The uni-directional switch sends "transactions". This permits the establishment of service nodes on the switch. Such a service node would be sent requests, would perform some action, and would reroute the request to the correct destination. The final destination could send the answer directly to the originator of the request. This cannot be done with a bidirectional switch.

While we considered the bidirectional switch during the early phases of this design, we were intrigued with the transaction switching nature of the uni-directional switch. The bidirectional switch has only recently come back into consideration and as a result, we are still considering its impacts on the structure of the switch and the rest of the system.

6.9 Speed Issues

For the Voice Funnel application an important speed issue is the bandwidth of a Butterfly Switch. This bandwidth is a function of five factors:

1. Switch thickness - The bandwidth is linearly related to how many bits move across the switch during one clock period.

2. Switch clock frequency - The bandwidth is linearly related to the switch clock frequency.

3. Control overhead - As each message consists of data bits and control bits, the bandwidth is related to the ratio of data bits to message bits (i.e., data bits plus control bits).

4. Memory bandwidth - At some operating point of the above four factors, the data cannot be written into or read from the local memory fast enough to keep the switch path busy and still permit the local processor an occasional access to its memory.

The switch clock frequency is limited by the time it takes to establish a connection between a switch node input and output port or by the time it takes to propagate the data to the next switch node. In very large switch configurations, the propagation time may be larger than the connection time. We have estimated the maximum clock rate using standard TTL Schottky MSI components. The worst case connection time is 77 nsec and the worst case propagation time is 40 nsec. Because of the 4 MHz maximum clock frequency of the microprocessor, a 12 MHz switch clock frequency seems to be a good choice.

Because not every switch clock period transfers data, the potential bandwidth of 48 MHz is further reduced. For the unidirectional switch, the table below shows the reduction in bandwidth due to the inclusion of control bits in each message for several data block sizes: (Base-4, 128 processors, thickness of 4).

Data Bits	Message Size	Effective Bandwidth (MHz)
16	96	8
32	112	13.7
64	144	21.3
256	336	36.6

As can be seen, data sent or retrieved from remote memory in multiple word blocks makes much better utilization of the Butterfly Switch.

6.10 Deadlocks

Consider a switch interface having a single input message buffer and a single output message buffer. The deadlock shown in Figure 6.10-1 can be imagined as resulting from the following scenario: 1) both Node A and Node B send memory "request" messages to each other simultaneously; 2) after these requests have been processed, but before the replies have been sent, requests from two other nodes (X and Y) arrive at the inputs. At this point, the "replies" in Node A and Node B cannot be sent because the receive buffers are occupied by the foreign requests. Yet, no further processing of requests can be done to free the receive buffers until the replies have been sent -- the system is locked up.

However, if we can assure that a reply message can always be sent and accepted, the system is free of deadlocks. This guiding

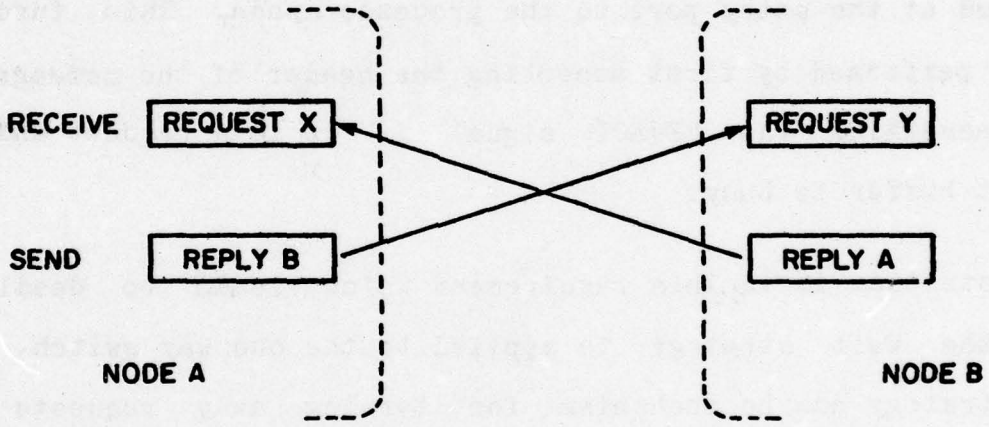


Figure 6.10-1 A Deadlock

principle will help us understand the requirements on the design of the switch and the switch interface which are necessary to assure deadlock-free operation. To validate these ideas, the simulations which we will be describing later have implemented these mechanisms and have not exhibited deadlocks.

If we are to assure that replies are always accepted, we must provide a mechanism which accepts them even if the receiver buffer is occupied. To assure this, we provide two buffers: one for requests and one for replies. This is not enough; we must also have a way to peek at each message and turn requests away when the request buffer is occupied so that replies can always be accepted at the entry port to the processor node. This function can be performed by first accepting the header of the message and then asserting the REJECT signal if it is a request and the request buffer is busy.

Note that it is this requirement which leads to deadlocks when the Wait strategy is applied to the one way switch. The Wait strategy has no mechanism for turning away requests and letting replies through.

If we are to assure the path of a reply, we must also assure that it can be transmitted in the first place. The straightforward way to do this is to provide two transmit buffers and allocate one for replies only. Then, when a request is rejected, before retransmitting the request, the transmitter should check the reply buffer and if a reply is present, send it instead of the request.

Finally, we are counting on randomness in the switch to assure that replies do not repeatedly encounter the same or other

request messages in the switch, thus also leading to deadlocks. This is a level of detail which we have not yet explored.

Interestingly, the bidirectional switch is free from all of these deadlocks since it pre-allocates the path through the switch which is to be used for the reply, and furthermore, the reply buffer is naturally independent of the incoming request buffer. This is one of the more attractive properties of the bidirectional switch.

6.11 Error Control

We have used the analogy of the Butterfly Switch as a communications network before. As we know, one of the important characteristics of a communications network is its handling of errors. We expect two things of such a network: 1) extremely low probability of undetected errors, and 2) automatic recovery from errors.

Although the Butterfly Switch is within a computer system, we should still expect errors to occur -- if not because of noise, then because of either intermittent or solid component failures. We have discussed the introduction of extra columns in the switch to provide extra paths which will permit operation once failed components have been identified. In order to detect errors when they occur, we expect to provide check bits on each transaction.

Errors in the data of a message will be detected by these check bits, but errors in addressing will not be. A simple way to detect addressing errors is to include the destination address in the checksum when the message is sent and to have the destination processor node include its own address when it verifies the checksum. Thus, if the message reaches the correct destination without error, the checksum will be correct. Otherwise, an error will be detected. This method has the advantage of avoiding the transmission of the destination address in the text of the message.

6.12 Flow Control

Another characteristic which we have come to expect of communications networks is a facility for flow control. This is necessary whenever we cannot guarantee that the receiver has sufficient resources to accept what the transmitter may wish to send. Our initial design for the switch interface implies a very structured buffering arrangement with few message format variations. As a result, our initial design would not need flow control in the switch.

6.13 Current Switch Design

In summary, we felt that the best switch design for the range of machines currently anticipated is as follows:

1. Conflict Resolution - We will use the "retreat" strategy.

2. Parallel Data Paths - Data parallelism of 4 bits (called a "nibble").

3. Switch Base - Base-4.

4. Extra Columns - None for a 16 input switch, perhaps one for a 64 or 256 input switch.

5. Depopulating Inputs - See "Partial Switches," below.

6. Partial Switches - If the number of inputs does not match a normal switch size, the next size switch will be used and the input ports of the switch depopulated to provide better performance.

7. Long/Short Messages - Both long messages (multiple word transfers) and short messages (single word transfers) will be supported. To keep the latency of short messages low, long messages will be broken into blocks of 16 words or less.

8. Bidirectional Switch Paths - We will be studying this variation in order to decide whether it is the best choice.

9. Speed - We expect the switch to clock at between 10 and 12 MHz for an effective bandwidth of 40 to 48 MHz per path. In a switch with 256 ports, this implies a maximum aggregate data rate of 10 to 12 gigahertz.

10. Deadlocks - We will avoid deadlocks by using the "retreat" strategy in the switch, and by designing the switch interface so that it can always accept a reply.

11. Error Control - The switch itself will not perform any error control, but the switch interfaces will use check bits to detect errors in data or routing.

12. Flow Control - None in the switch. The sender and receiver will be designed so that the transactions do not need flow control.

7. Switch Performance

The Butterfly Switch in a rather complex system. Before using it in the Butterfly Multiprocessor, we have simulated its performance. Our intention is as much to detect any serious weakness in the design as it is to learn about the fundamental characteristics of this interesting structure. At the same time, we have used simulations to verify that the particular design decisions we have made are reasonable and do not have a more significant effect on the performance of the system than was expected.

It is possible that the behavior of a Butterfly Switch under load could be studied with classical analytical methods, but such analysis is, at the least, very difficult. The system appears to be an open network of M/D/K queues [KLEI 75], also known as a "link system" in telephone terminology [SYSK 60]. Although individual M/D/K queues can be analyzed, to the best of our knowledge there are no techniques which generalize to networks of these systems.

Open networks of M/M/K are solvable, at least theoretically, although in practice they involve gargantuan computations. Since M/M/K serves customers faster than M/D/K under similar conditions, this could give an upper bound. We have not pursued this particular investigation.

It is possible, however, to simulate the Butterfly Switch and to observe its behavior with differing policies. We have done this and present the results in this section. The results are presented along the lines of the variations described above.

7.1 The Simulator

The simulations described here are what might be called full system simulations. The objective has been not only to simulate the Butterfly Switch with some artificial traffic load, but rather to simulate the entire environment of the Butterfly Switch as well. This has the advantage of permitting a more direct view of the impact of design choices on the performance of the Butterfly Multiprocessor. It also has the disadvantage of limiting the information about the inner workings of the switch itself and further complicates efforts to match analysis and simulation. We have nonetheless chosen a system simulation and have used the system efficiency as the measure of switch design decisions.

In this simulation, the Butterfly Switch is modeled as a collection of finite state machines, a Markov model, each representing a component capable of "independent" action. The processing node has four finite state machines: the processor itself, the local memory, the switch interface transmitter, and the switch interface receiver. Each transaction (message) is

itself a finite state machine. All machines change state synchronously at time units called "ticks", which here represent the rate at which data moves through the switch, assumed to be 83 nsec.

The processor alternates between the "computing" state and some form of memory access. The number of cycles spent computing is determined from a fixed distribution based on the instruction statistics of the Z8000.

After a compute cycle, a memory cycle is executed. This memory cycle is either an access to a local memory location, or to a remote memory location. The probability of such a remote memory reference, Prob(Remote Reference), is the independent variable in these simulations. If the reference is to local memory, the simulator sees if the memory is busy (it may also be accessed by the switch interface). If not, then it is made busy for a number of ticks which represents the memory cycle time after which the processor can begin a new computing cycle. If the memory was busy, the processor must wait until it becomes idle before beginning the local memory cycle.

When the reference is to remote memory, the switch interface transmitter must be used. If the transmitter is in use, such as when it is sending a reply, the processor must wait until it is available. When it becomes available, the remote memory request

message can be transmitted. Two parameters of the message which are important at this point are its destination and its length. The destination is chosen randomly from al.. ıossible processing nodes except itself. The length depends on whether it is a read or write. The probability of a read is assumed to be 0.70.

Once transmitted, the message proceeds through the switch, taking two ticks for each column in the forward direction. If another message is encountered that is using a path which this message needs, a conflict is noted and an appropriate response is taken (wait or retreat).

When the message reaches its destination, it is handled by the switch interface receiver. The receiver notes that it is a request type message and makes the corresponding local memory request, waiting, of course, if the memory is busy. When the cycle completes, the receiver asks the switch interface transmitter to send the request. If the transmitter is busy (sending something for the processor, for example), the receiver must wait. The answering message is treated much the same way as the request, save only that its destination is pre-determined -- the source of the requesting message -- and its length now is different.

When the answer arrives at the switch interface receiver, we note that it is an answer. The processor has all this time been

in an "answer wait" state; we can now advance it to the next computing cycle.

As discussed previously, the system as described can (and does!) deadlock. We must fix this at both the switch interface receiver and transmitter. The question at the receiver concerns the time when the receiver finite state machine is waiting for the transmitter to become idle. When in this state, the receiver should reject any other incoming message -- it has no place to put it. However, if the incoming message is an answer (as opposed to a request), then the receiver SHOULD accept the message and let the processor move into its next computing cycle.

The switch interface transmitter has been modified so that one request message and one response message can be awaiting transmission at the same time. When a message encounters a conflict in the network or a rejection from the destination, the transmitter should alternate between a queued answer and a queued request. Thus if a message retreats back to the originating switch interface, the transmitter sees if there is another message waiting and sends it instead.

The measure of performance of the system, the dependent variable in the graphs, is the System Efficiency. This is a measure of the fraction of a processor that is doing useful work. It is computed by allocating a "credit" for each tick in which

the processor is either computing or actually accessing either local or global memory. Credits are not given when messages are flowing through the switch or when a processor is waiting for some resource, either memory or the switch interface. The total number of credits is then divided by the total number of ticks in the simulation and normalized by the number of processors to give the system efficiency. This figure thus represents the degradation of the system which arises from delays and contention. The system would be operating perfectly at a system efficiency of 1.

A problem with simulating something as complex as a Butterfly Switch is that there are many parameters, all of which can be varied. Our approach has been to pick defaults for each of the parameters and vary one at a time while keeping the others fixed at the default value. Here are some of the key parameters and their defaults:

Number of processors	64
Base of switch	4
Conflict strategy	retreat
Length of write request	14 nibbles
Length of all other messages	11 nibbles
Prob(Remote Reference)	0.10
Addressing	random

7.2 Performance of a Butterfly Switch

The results of the simulations described above are presented in the four graphs in this section. These graphs show that:

1. The performance of the Butterfly Switch we have selected compares favorably with a similar crossbar.
2. The selection of a base-4 switch over a base-2 or base-8 switch represents a good compromise between the performance and size of these switch nodes.
3. The selection of a thickness of 4 is also a good choice since it is a manageable amount of parallelism and results in much improvement in performance over a thickness of 2. Selection of twice as much thickness improves the performance of the switch but not enough to offset the extra bulk in the switch.
4. The number of processor nodes which are attached to the switch has a large impact on the performance of the switch for some remote reference probabilities.

Some explanation of the graphs is in order. In each graph we are plotting the System Efficiency versus the Probability of a Remote Reference. This probability is plotted logarithmically, centered around a probability of 0.1. In each graph, the solid line represents the same switch design parameters, a base-4 Butterfly Switch with thickness of 4 and 64 ports. This line is used as the basis of comparisons.

To put these probabilities in perspective, notice that if every reference (instructions and data) the processor makes is to remote memory, then the probability is unity. If the

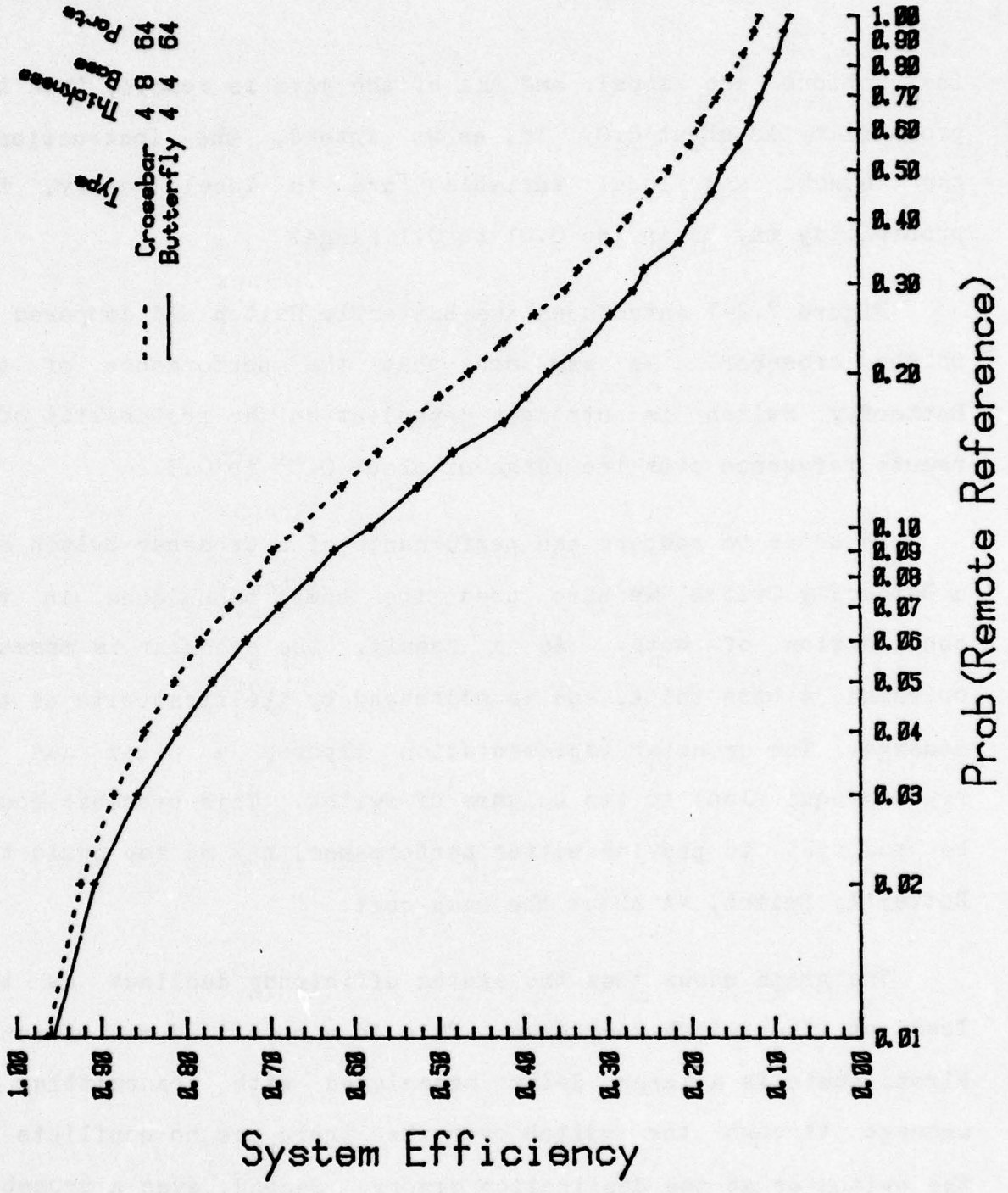


Figure 7.2-1 Butterfly Switch vs Crossbar

instructions are local and all of the data is remote, then the probability is about 0.2. If, as we intend, the instructions, the stack, and local variables are in local memory, the probability may be in the 0.01 to 0.1 range.

Figure 7.2-1 introduces the Butterfly Switch and compares it to the crossbar. We can see that the performance of the Butterfly Switch is strongly dependent on the probability of a remote reference over the range of about 0.03 to 0.3.

In order to compare the performance of a Crossbar Switch and a Butterfly Switch, we have used the same techniques in the construction of both. As a result, the crossbar is message oriented, 4 bits thick, and is addressed by the first bits of the message. The crossbar implementation imposes a delay due to routing equivalent to two columns of switch. This crossbar could be modified to provide better performance, but so too could the Butterfly Switch, at about the same cost.

The graph shows that the system efficiency declines as the load on the switch increases. This is a result of two effects. First, there is a large delay associated with transmitting a message through the switch even when there are no conflicts in the switch or at the destination memory. Second, even a crossbar switch is affected by contention for the remote memory. The Butterfly Switch adds to these effects the possibility of

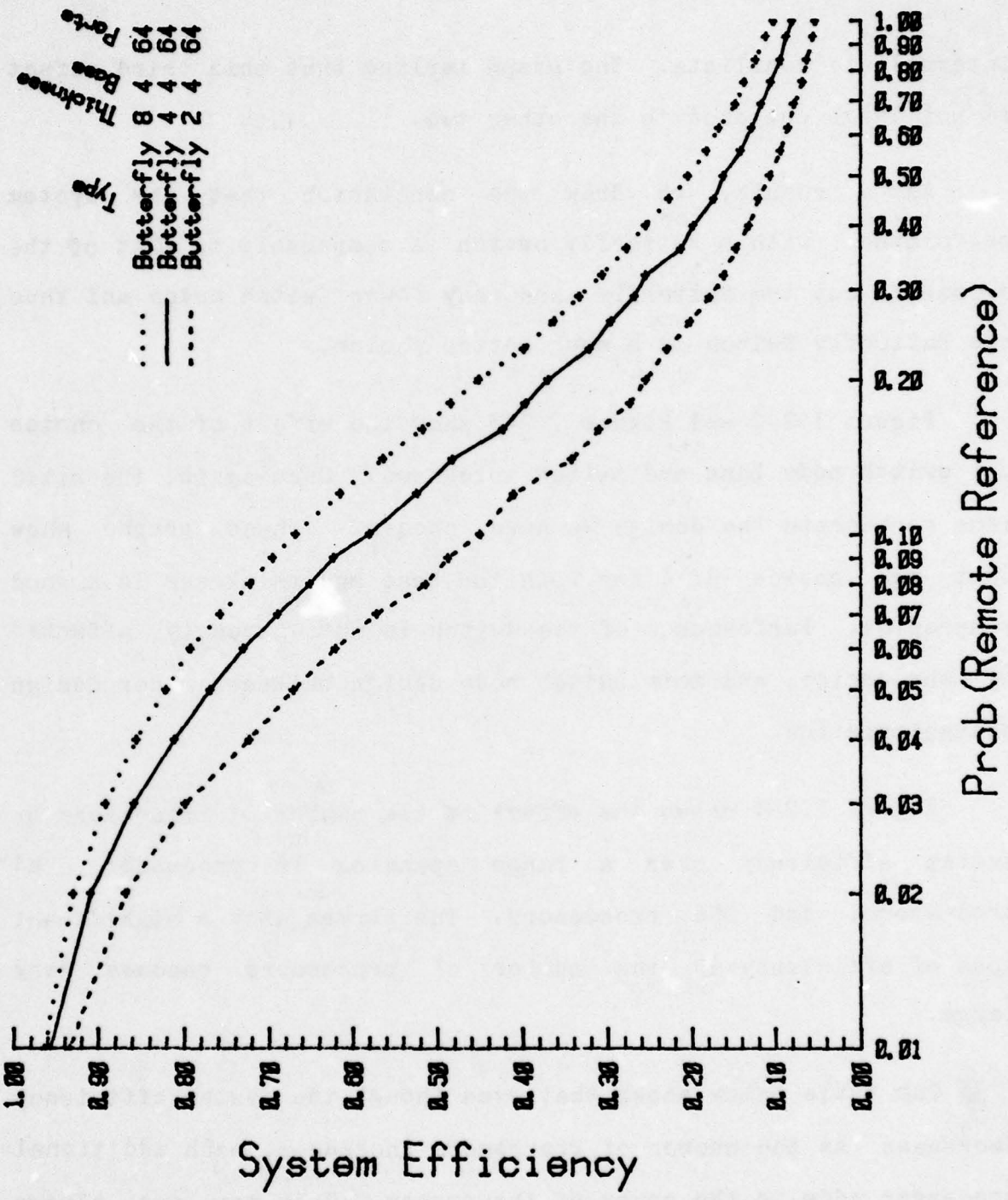


Figure 7.2-2 Effect of Data Parallelism

intermediate conflicts. The graph implies that this third effect is not large compared to the other two.

As a result, we draw the conclusion that the system performance with a Butterfly Switch is comparable to that of the crossbar, but the Butterfly uses many fewer switch nodes and thus the Butterfly Switch is a much better choice.

Figure 7.2-2 and Figure 7.2-3 show the effect of the choice of switch node base and switch thickness. Once again, the solid line represents the design we have chosen. These graphs show that the choice of 4 for both the base and thickness is a good compromise. Performance of the switch is not strongly affected by the choice, and this switch node design balances other design characteristics.

Figure 7.2-4 shows the effect of the number of processors on system efficiency over a range spanning 16 processors, 64 processors, and 256 processors. The curves show a significant loss of efficiency as the number of processors becomes very large.

The table below shows that even though the system efficiency decreases as the number of processers increases, each additional processor adds to the power of the system. This may not always be true or it may be that this structure reaches a point where adding another processor becomes no longer cost effective. For

the range of systems we are now considering, this is not yet a problem. For example, the table below shows the effective number of processors when the probability of a remote reference is 0.01 and 0.1:

Actual	Effective Processors (Prob=.01)	(Prob=.1)
16	16	12
64	62	37
256	230	78

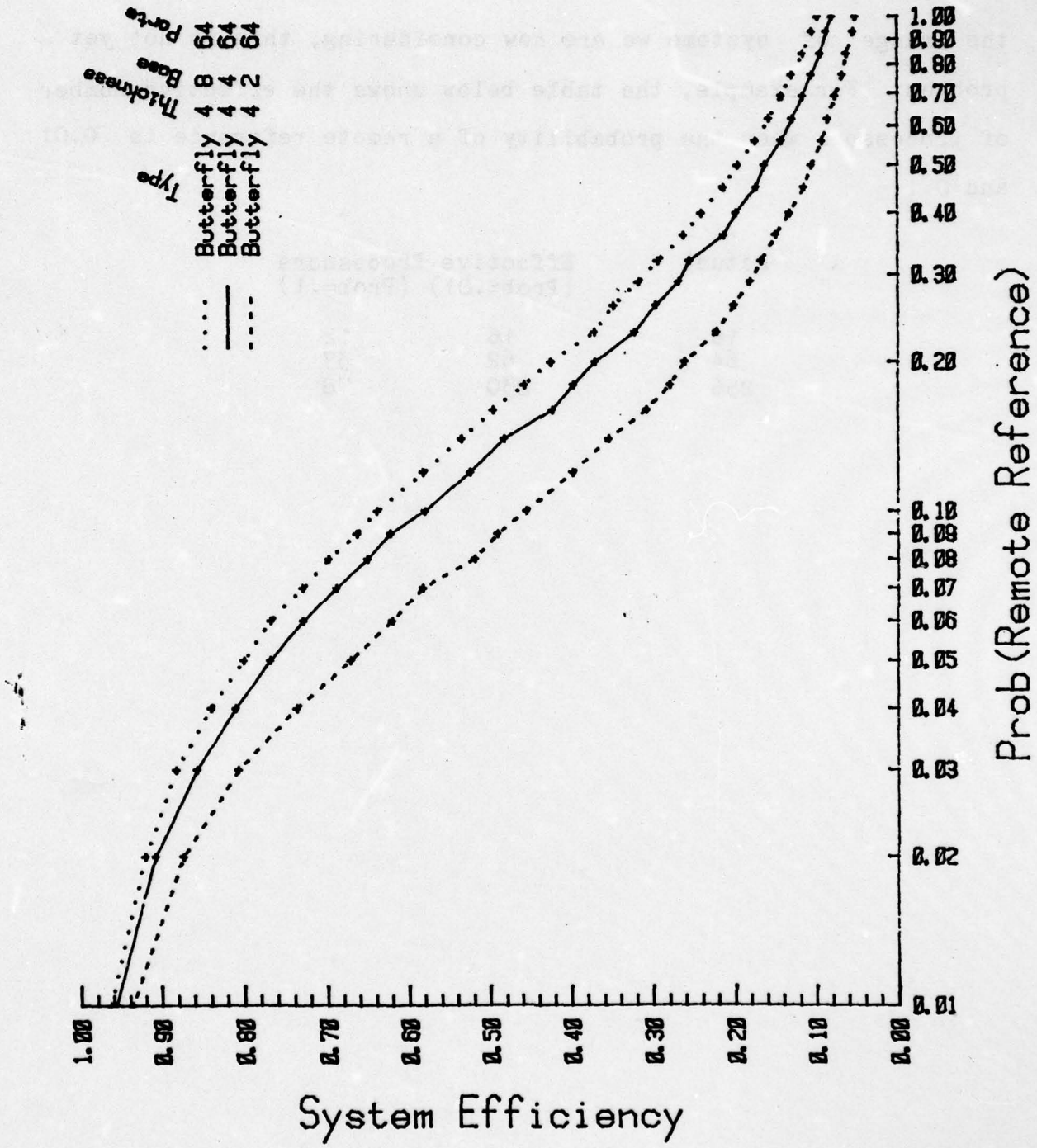


Figure 7.2-3 Effect of Different Bases

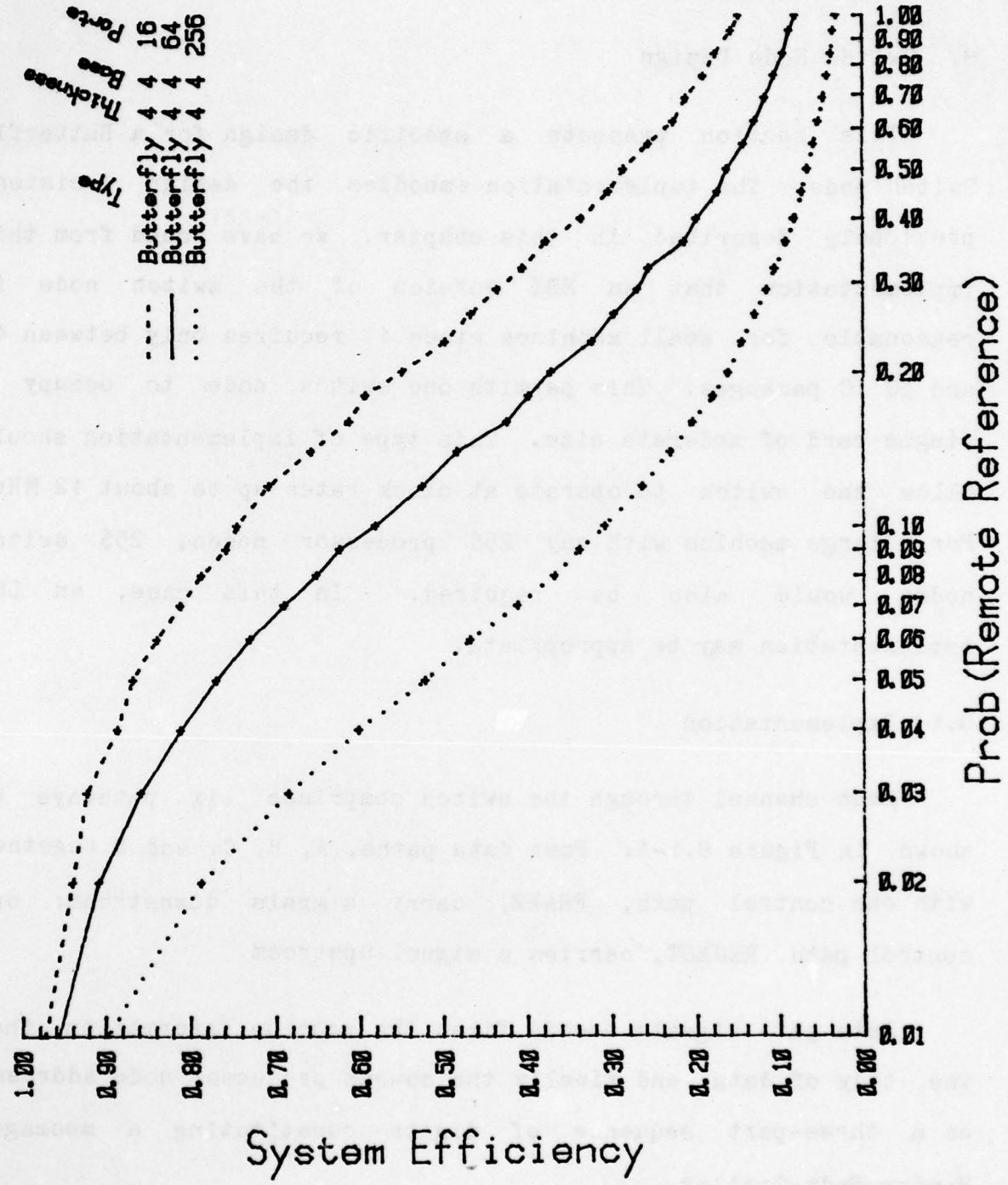


Figure 7.2-4 Effect of Number of Ports

8. Switch Node Design

This section presents a specific design for a Butterfly Switch node. The implementation embodies the design decisions previously described in this chapter. We have found from this implementation that an MSI version of the switch node is reasonable for small machines since it requires only between 40 and 60 IC packages. This permits one switch node to occupy a single card of moderate size. This type of implementation should allow the switch to operate at clock rates up to about 12 MHz. For a large machine with say 256 processor nodes, 256 switch nodes would also be required. In this case, an LSI implementation may be appropriate.

8.1 Implementation

Each channel through the switch comprises six pathways as shown in Figure 8.1-1. Four data paths, A, B, C, and D together with one control path, FRAME, carry signals downstream; one control path, REJECT, carries a signal upstream.

Data path signals encode first the routing information, then the body of data, and finally the source processor node address, as a three-part sequence of digits constituting a message: Header-Body-Trailer.

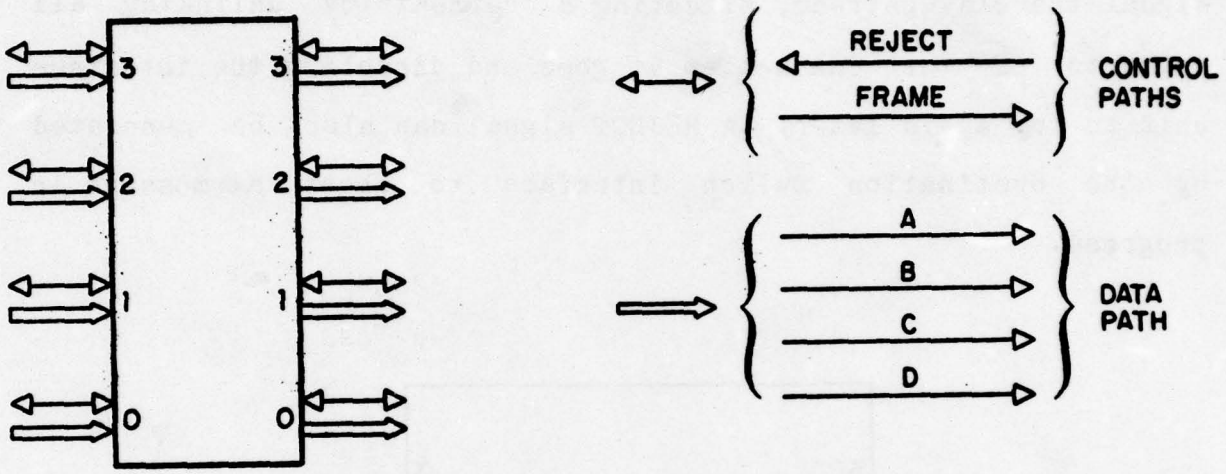


Figure 8.1-1 Base-4 Switch Node I/O

The signal FRAME accompanies each message and serves to define the Head and Tail of a message. It is generated by the interface unit to this node's left. Head and Tail indications are used by the switch node in routing and in source address code generation, and are passed on in the form of a new Frame signal to the interface unit to this node's right to delimit the transmission. As the Head enters a node a crosspoint link is formed; as the Tail leaves the node, the link is broken.

The signal REJECT is generated by a node that is not free to establish the required link when the message Head appears. The signal travels upstream, effecting a "retreat" by unlinking all segments of the channel as it goes and directing the interface unit to try again later. A REJECT signal can also be generated by the destination switch interface to abort a message in progress.

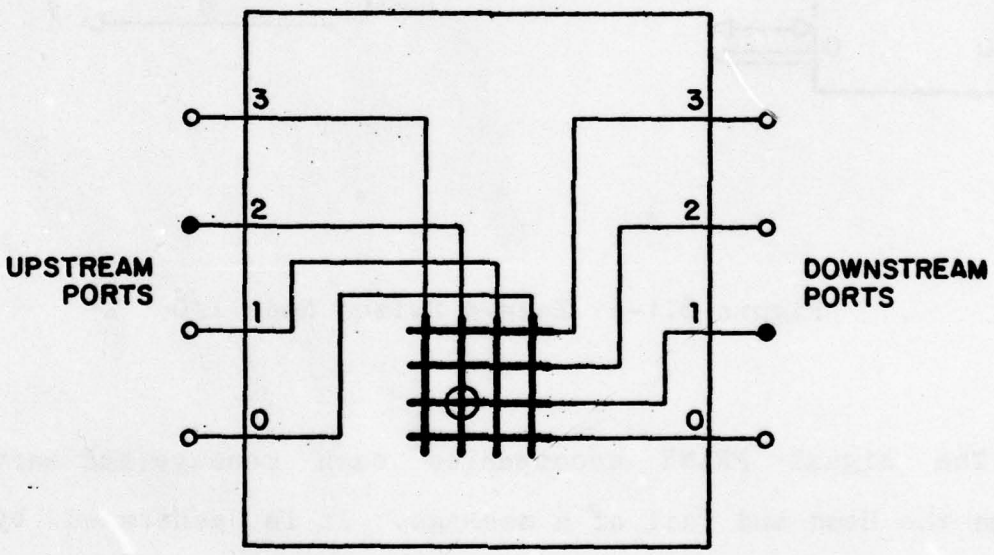


Figure 8.1-2 Base-4 Switch Node

The base-4 switch node incorporates the functional equivalent of a 4 X 4 crossbar switch as shown in Figure 8.1-2.

Within the node, a channel link is uniquely identified by a pair of base-4 digits u and d (for upstream and downstream). We will use the notation [u - d], where u,d = 0,1,2,3, to designate the link from port u to port d and will refer to u and d as the link indices.

Figure 8.1-3 shows an MSI implementation of that part of the 4 X 4 switch node involved in the link [2-1]. (The structure of each of the other links is identical to that shown.) The crossbar elements are 4-to-1 Multiplexers with the data input port determined by selection code inputs, A and B, and output gated by an input signal G. A gated Bid Decoder produces a bid signal, 2 BID 1, on address input A2 of a programmable read-only memory (PROM) array which implements the Bid Priority and Crossbar Assignment Logic.

The design has been modularized in both upstream (vertical) and downstream (horizontal) planes of the crossbar array. Since multiplexers are not inherently bidirectional, upstream-oriented multiplexer, 2XR, and downstream-oriented multiplexers, XF1 and XA1 through XD1, are required in the planes shown.

Logic nets on the buffered FRAME pathway detect the head and tail of the message. The Head signal and the requisite downstream link index (currently encoded on data paths A and B during the leading digit of the message) are sent to the Bid

Priority and Crossbar Assignment Logic block. Taken together, Head and index constitute a bid for the establishment of a crossbar link to the node's indicated downstream port.

If the port is available and if no other message having higher priority (according to some ordering scheme) is simultaneously bidding for use of the same port, then the link is established; the assigned connection code is set into flip-flops and held for the duration of the message, and from there fed to the crossbar switch to establish the connection. The Data and FRAME signals are sent through to the downstream port. The Frame signal is retarded by one clock tick with respect to the Data, thus in effect discarding the old leading digit of the Header. This signal announces to the Bid Logic that a message is in transit and that the port is therefore unavailable to other bidders.

If the port is unavailable at bid-time or if a higher priority bid is simultaneously being made, a Local Reject signal is returned to the hapless bidder's channel logic, where it resets the Frame flip-flops and continues upstream in retreat, becoming a Downstream Reject signal at the next node upstream. Here it again clears the Frame flip-flops and continues upstream, and so on, taking one clock tick per node.

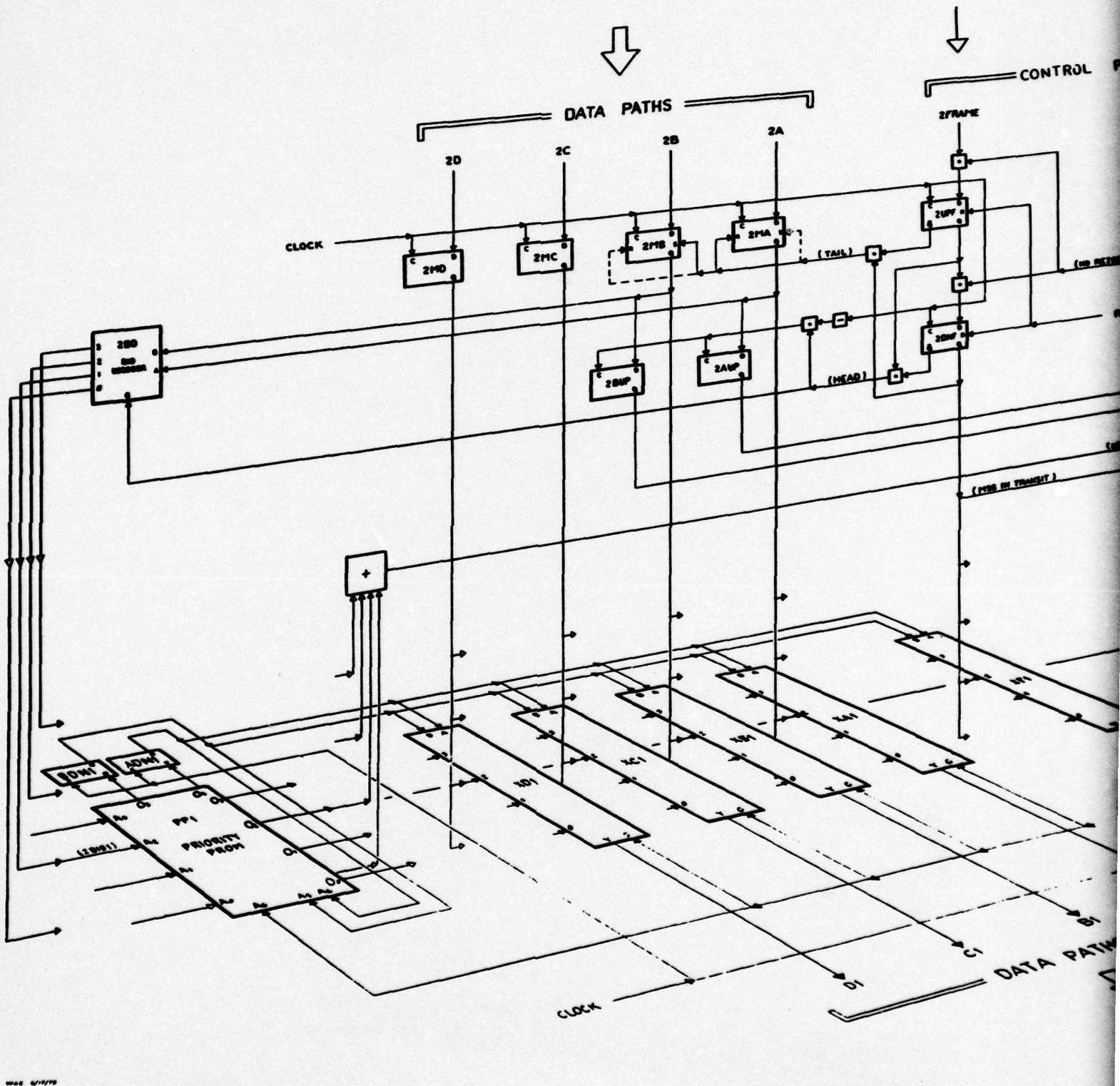
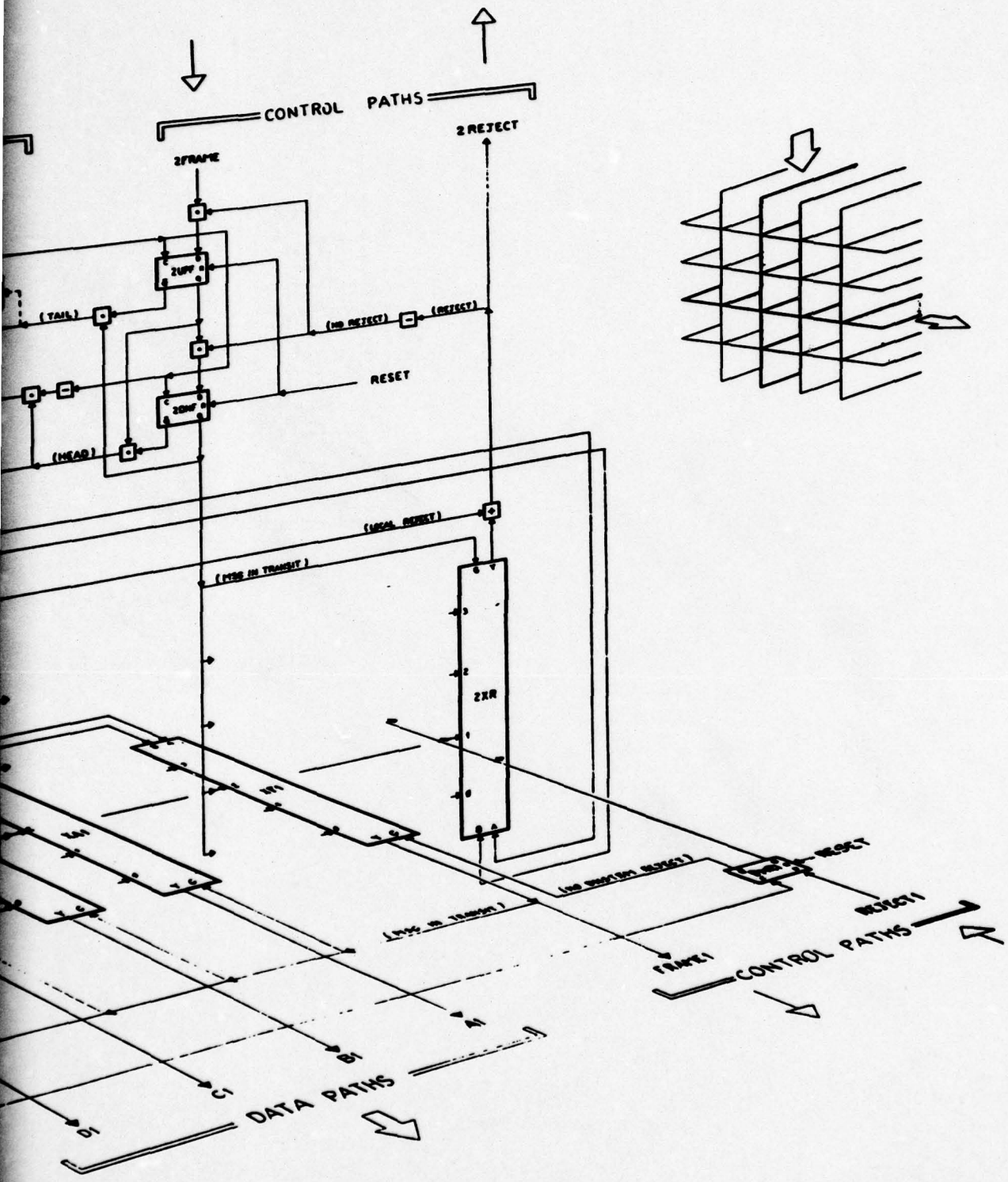


Figure 8.1-3 Base-4 Switch Node

1


CONTROL PATHS
2REJECT
2FRAME
2UPF
(TAIL)
(NO REJECT)
(REJECT)
RESET
2DNF
(HEAD)
(LOCAL REJECT)
(MSG IN TRANSIT)
2XR
(MSG IN TRANSIT)
(NO DOWNSTREAM REJECT)
REJECT
REJECT1
FRAME1
CONTROL PATHS
A1
B1
C1
D1
DATA PATHS

Retarding the downstream FRAME with respect to the Data also has the effect of creating a new digit position at the end of the message trailer. The signal, TAIL, causes this position to be filled with the upstream link index which it sets (by fixed wiring) into the Data flip-flops on the B and A pathways.

The reader will see that the Frame signal must pass through two flip-flop stages (2UPF and 2DNF) whereas the Data signals pass through only one (flip-flops 2MA through 2MD). This difference produces the required relative retardation of the Frame signal. The messages' head and tail are easily detected in the combined states of 2UPF and 2DNF. HEAD enables the bid as well as the mid-tick capture of the downstream link index in flip-flops 2BUP and 2AUP; TAIL sets the upstream link index code into flip-flops 2MB and 2MA (the dashed lines indicating the wiring possibilities in other planes).

The Bid Priority and Assignment Logic is modularized in both upstream and downstream planes. A 128 X 6 PROM array stores the required code patterns: downstream multiplexer Selection Codes, which are sent to flip-flops BDN1 and ADN1, and Local Reject Codes, which are returned to the upstream planes. The address input signal on A6, MSG IN TRANSIT (=FRAME1), selects one of two major tables stored in the PROM. One table stores Local Reject Codes that result in rejection of all new bidders while a message is in transit. The second table permits a new link selection to

be made if a bid is received, and specifies the order in which simultaneous bids are accepted.

An especially nice feature is provided by feeding back the previous selection code from the flip-flops BDN1 and ADN1 to PROM address inputs A5 and A4, thereby further partitioning the stored patterns into four sub-tables. Once a selection has been made on the basis of codes stored in sub-table 1, the next selection will be made on the basis of sub-table 2, and so on in "round-robin" fashion. Priority order is determined by the stored patterns and differs from sub-table to sub-table so as to assure balanced servicing of competitive bidders under conditions of heavy traffic.

Although it simplifies the priority logic to a considerable extent, the PROM array constitutes the slowest element of the node. Nonetheless, it appears that the node, if its logic is implemented in Schottky TTL form (which requires about 45 integrated circuit packages) will operate at 12 MHz.

8.2 LSI Implementation

The implementation described above is reasonable for small switches, as each switch node can be constructed on a small card. However, the class of multiprocessors described in this document is not limited to a small number of processing nodes. The following table shows the number of switch nodes required for several "natural" switch sizes.

Ports	Switch Nodes
4	1
16	8
64	48
256	256
1024	1280
4096	6144

As we can see, the number of switch nodes becomes equal to and even exceeds the number of processing nodes in large machines. The processor nodes are already about as highly integrated as the current technology permits, but the switch nodes are not. The two deliverable machines under this contract are small enough that an LSI implementation is not necessary. However, integration of the switch node would permit the expansion of this machine over a larger range.

A 4 X 4 node with 6 paths per channel has 48 signal pathways. These, together with paths for electrical power, clock, and reset signals lead to a requirement for at least 52 paths per node. A fully integrated structure would therefore fit into a 64-pin package, currently one of the industry standards.

Integrated circuit technology permits much more complex logic than the above functional design requires. For example, one might easily include buffer memories to improve switch performance. On the other hand, an implementation of the design in currently available medium- to small-scale integrated circuits is limited in complexity but not seriously limited in the number

of pins per package. We have tried to keep a balance between these contrasting considerations in the design.

9. Summary

The Butterfly Switch is attractive for use in a multiprocessor. The switch is a mid-point between completely connected switches and singly connected switches. Its performance is comparable to that of the crossbar at a significant reduction in the size of the switch.

We have found that a switch built from switch nodes which are themselves 4 X 4 crossbar switches with data paths 4 bits wide is a significant improvement over the originally proposed design. We have also found that the "retreat" strategy should be used in the switch because it gives better performance than the original "wait" strategy and is free of deadlocks.

Two areas require further study: 1) the bidirectional switch design, and 2) the effect of contention at the destination as the number of processors increases. We expect to continue to examine these areas and others as the design progresses.

10. References

[ABRA 69] N. Abramson, "The ALOHA System -- Another Alternative for Computer Communications," AFIPS Conference Proceedings 37, 1969.

[BATC 68] K.E. Batcher, "Sorting Networks and their Applications," AFIPS Conference Proceedings 32, 1968.

[BHAN 77] D.P. Bhandarkar, "Some Performance Issues in Multiprocessor System Design," IEEE Transactions on Computers, May 1977.

[CHEN 74] R.C. Chen, Bus Communications Systems, Carnegie-Mellon University Report No. PB-235 897, January 1974.

[DENN 75] J.B. Dennis, "Packet Communication Architecture," Project MAC Computation Structures Group Memo #130, August 1975.

[FARB 72] D. Farber, "Data Ring Oriented Computer Networks," in Computer Networks, R. Rustin (ed.), Prentice-Hall, 1972.

[FLYN 72] M.J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Transaction on Computers, Vol. C-21, No. 9, September 1972.

[HEAR 73] F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," AFIPS Conference Proceedings 42, June 1973.

[KLEI 75] L. Kleinrock, "Queueing Systems. Volume I: Theory," John Wiley & Sons, N.Y., 1975.

[LANG 76] Tomas Lang, "Interconnections between Processors and Memory Modules Using the Shuffle Exchange Network," IEEE Computer Transactions, May 1976.

[LAWR 73] D. Lawrie, "Memory-Processor Connection Networks," Report No. UIUCDCS-R-73-557, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, February 1973.

[LIPO 77] G.J. Lipovski and A.Tripathi, "A Reconfigurable Varistructure Array Processor," Proceedings of the 1977 International Conference on Parallel Processing, August 1977.

[MANN 76] W.F. Mann, S.M. Ornstein, and M.F. Kraley, "A Network-Oriented Multiprocessor Front-End Handling Many Hosts and Hundreds of Terminals," AFIPS Conference Proceedings 45, June 1976.

[METC 75] R.M. Metcalfe, D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Xerox Palo Alto Research Center Report No. CSL 75-7, November 1975.

[NSC 77] "Systems Description: Series A Network Adapters," Network Systems Corp. Publication No. A01-000-01, 1977.

[ORNS 75] S.M. Ornstein, W.R. Crowther, M.F. Kraley, R.D. Bressler, A. Michel, and F.E. Heart, "Pluribus -- A Reliable Multiprocessor," AFIPS Conference Proceedings 44, May 1975.

[PEAS 68] M. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," JACM, April 1968.

[PEAS 77] M. Pease, "The Indirect Binary n-Cube Microprocessor Array," IEEE Transactions on Computers, May 1977.

[STON 71] H. Stone, "Parallel Processing with a Perfect Shuffle," IEEE Computer Transactions, February 1971.

[STON 72] H. Stone, "Dynamic Memories with Enhanced Data Access," IEEE Computer Transactions, April 1972.

[SWAN 75] R.J. Swan, S.H. Fuller, D.P. Siewiorek, "The Structure and Architecture of Cm*: A Modular, Multi-Microprocessor," in Computer Science Research Review, 1975-1976, Carnegie Mellon University, p. 32.

[SYSK 60] R. Syski, "Introduction to Congestion Theory in Telephone Systems," Automatic Telephone & Electric Co., Oliver & Boyd, 1960.

[WU 78] C. Wu and T. Feng, "Routing Techniques for a Class of Multistage Interconnection Networks," Proceedings of the 1978 International Conference on Parallel Processing, August 1978.

[WULF 72] W.A. Wulf, C.G. Bell, "C.mmp -- A Multi-Mini Processor," AFIPS Conference Proceedings 41, 1972.

Chapter IV: Processor Node

Chapter IV: Processor Node

Contents

Chapter IV: Processor Node

1. Introduction

Although we have concentrated much of our effort on the design of the Butterfly Switch, other components are needed to implement a Voice Funnel. These components are grouped together to form what are called processor nodes. The processor nodes support the Voice Funnel's I/O devices and connect to the Butterfly Switch. In this chapter, we present the current design of a processor node.

The rest of this chapter is divided into two sections. Section 2 discusses the elements of the processor node and how they interconnect. Section 3 presents a brief performance evaluation of the CPU used in the processor node.

2. Processor Node Elements

The processor node consists of five elements: the processor and its extension peripherals, I/O interfaces, switch message receiver, switch message transmitter, and memory. Their interconnections are shown in Figure 2-1.

The elements of the processor node execute many simultaneous operations. As an example, the following operations could be executing simultaneously:

1. A serial communications channel interface is assembling channel bits into a byte.
2. Another channel interface is reading the next byte to be sent from local memory.
3. The switch message receiver is receiving a message from a remote processor.
4. The switch message transmitter is retransmitting a message because of a previous conflict in the switch.
5. The processor is adding two numbers.

For high throughput environments (such as the Voice Funnel), multiple concurrent operations are normal and desirable. However, concurrent operations require parallelism in the hardware, increasing the cost and complexity of the processor node. Fortunately, advances in IC functionality make operation concurrency much simpler than in the past.

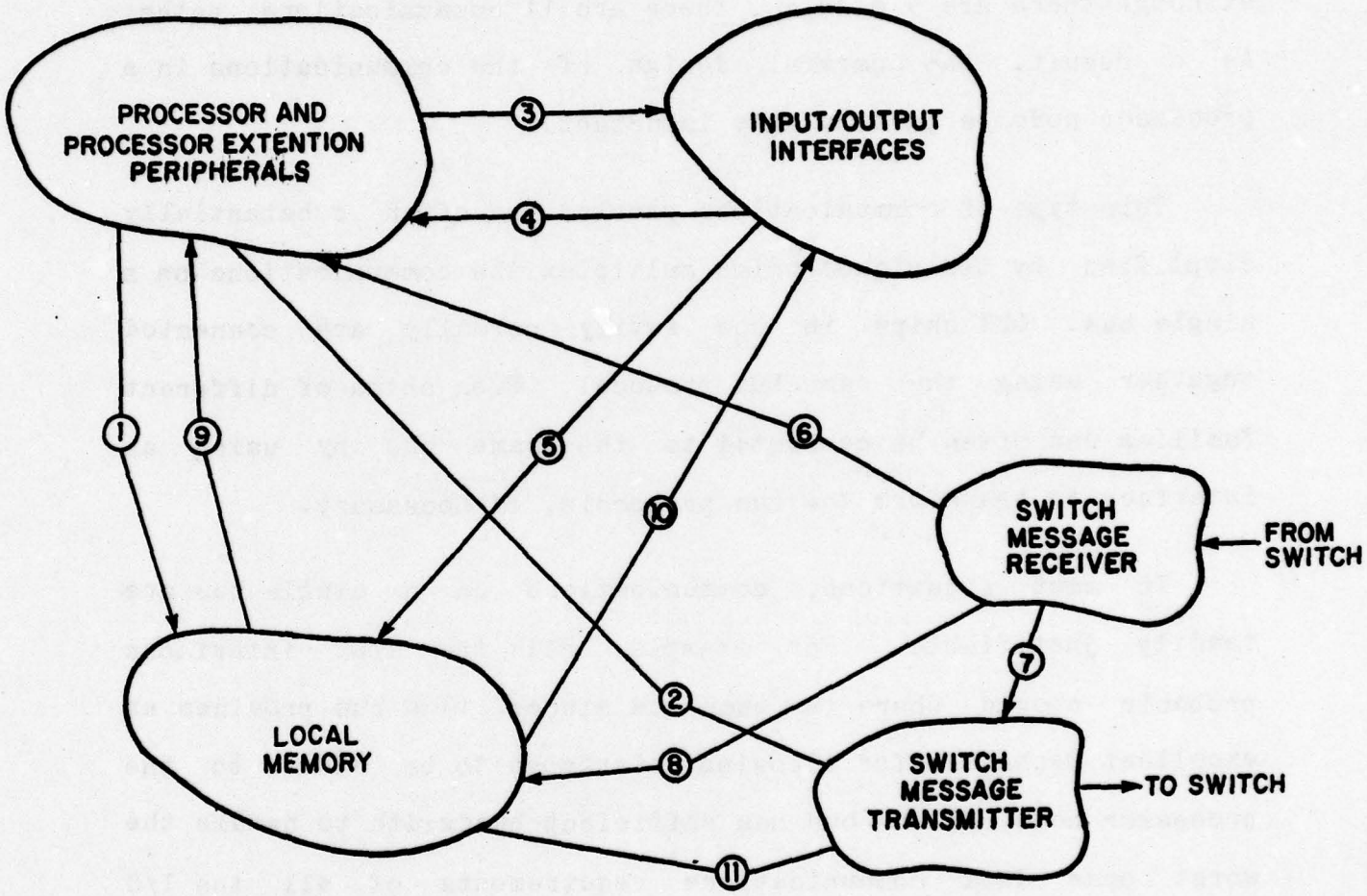


Figure 2-1 Processor Node Elements

An important design issue in the processor node is how to provide communications paths between the elements of the processor node. If we look at the figure again, we see that although there are 5 modules, there are 11 communications paths. As a result, the careful design of the communications in a processor node is particularly important.

This type of communications problem is often substantially simplified by techniques which multiplex the communications on a single bus. LSI chips in one family normally are connected together using the same bus protocol. Even chips of different families can often be connected to the same bus by using an interface to transform the bus protocols, if necessary.

In most situations, communications on a single bus are readily justifiable. For example, all the I/O interfaces probably should share the same bus since: 1) a bus provides an excellent mechanism for allowing interfaces to be added to the processor node, 2) the bus has sufficient bandwidth to handle the worst case data communications requirements of all the I/O interfaces connected to it, and 3) current I/O devices are bus-oriented.

Normally, the processor is simply placed on the same bus as the I/O devices. In the processor node for the Butterfly Multiprocessor, however, this is not possible since when the

processor gains exclusive control of the bus for the duration of a remote memory access (which could last hundreds of microseconds), I/O interfaces would be blocked and might not be able to meet the latency requirements imposed by real-time events. The easiest way to solve this problem is to provide a means of disconnecting the processor from the bus during this access and reconnecting it when the access is complete. This allows the other elements in the processor node to use the bus in the meantime.

Buses also have bandwidth limitations. Thus the switch message receiver, switch message transmitter, and local memory should not be connected to the relatively low speed I/O bus because they are high speed elements.

Before discussing the element interconnection scheme to be used in the processor node, it is useful to enumerate the element intercommunications that are required. The numerals match those in the figure.

1. Processor to local memory (e.g., the processor loads data into a local memory location).
2. Processor to switch message transmitter (e.g., the processor accesses memory in another processor node).
3. Processor to I/O interfaces (e.g., the processor loads data into an I/O device).
4. I/O interface to processor (e.g., I/O interface interrupts the processor or the processor reads data from an I/O device).

5. I/O interface to local memory (e.g., I/O interface loads incoming data into memory via a direct memory access).

6. Switch message receiver to processor (e.g., the processor reads memory in another processor node).

7. Switch memory receiver to switch message transmitter (e.g., the source address from the received message must be included in the reply message).

8. Switch message receiver to local memory (e.g., a remote processor has requested a word from this local memory).

9. Local memory to processor (e.g., the processor reads a location in local memory).

10. Local memory to I/O interface (e.g., I/O interface reads a location in local memory via direct memory access).

11. Local memory to switch message transmitter (e.g., the processor initiates a block transfer from local memory to memory in another processor node).

The remaining possible interconnections are either architecturally inconsistent or are not allowed. For example, a local memory to switch message receiver path is nonsense since the switch message receiver cannot be a data sink. The I/O interface to switch message transmitter path is not allowed because the potentially long message transmit delay could cause incoming data to be lost.

In addition to providing connectivity between the various processor node elements, it may be necessary to establish several simultaneous data paths. These extra paths allow us to trade complexity and cost (to implement the special bus drivers and control for example) for performance. The value of this approach is determined by the intercommunications statistics.

Although we do not know the statistics for the processor node, it seems that the performance improvement resulting from multiple communication paths is not worth the extra cost and complexity. We have therefore adopted a single high speed bus as shown in Figure 2-2. The bus is 16 bits wide and operates synchronously at the switch clock frequency.

2.1 The Processor

The processor we have selected for use in the Voice Funnel is Zilog's Z8000. This machine has been announced with delivery expected before the summer of 1979. We have selected it because 1) it is a 16-bit microprocessor, 2) it has a reasonably high instruction execution rate, 3) it can manipulate a 23-bit virtual address space, and 4) it will have a compatible memory management device.

2.1.1 Processor Candidates

There are in general three categories of candidates for the processor in the Voice Funnel:

1. Commercial minicomputer
2. Commercial microprocessor
3. Custom microprogrammed processor

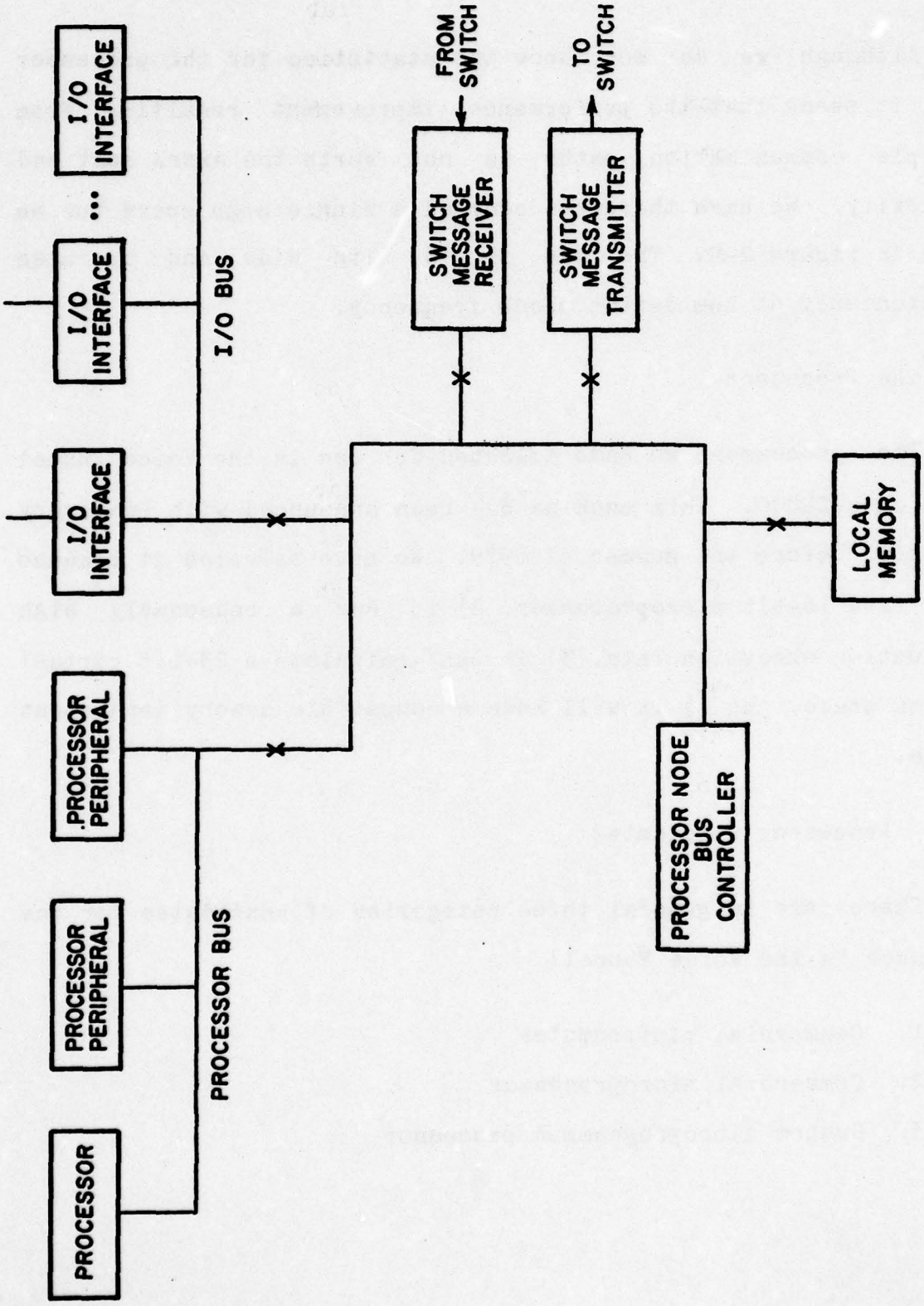


Figure 2-2 Processor Node

The advantage of a commercial minicomputer is the commitment on the part of the manufacturer: it is likely that compatible processors which are more powerful and/or lower in cost will be produced in the future. Furthermore, a complete line of compatible equipment and software is probably available. For microprocessors this has been true only over a very limited range. For example, the 8080 is now available at lower cost and in much faster versions, but the new architecture, the 8086, is only awkwardly compatible with the 8080. Similarly, Zilog with their Z8000 and Motorola with their M68000 have developed new architectures for their new products. A custom processor is even worse in this respect, since every product enhancement must be supplied by local effort.

The advantages of a commercial microprocessor are its small size, low cost, and multiple sourcing. The first two are particularly important since we expect to have many processors in our system, which puts a premium on a processor which has a high performance-to-size ratio as well as a high performance-to-cost ratio. We would also like to have a machine which adapts well to technological advancement. This is, after all, the age of LSI. It is clear that these technologies offer so much that it is hard to avoid taking advantage of them. The pattern of multiple sourcing is simply an added benefit. Multiple sourcing is not common in the arena of commercial minicomputers; the

vulnerability that results from a single supplier can be serious. A custom machine can be of moderate cost (between a commercial mini and a micro) but its size is much larger than a micro.

The primary advantage of a custom microprogrammed processor is that it can be designed to fit well into the overall structure of the machine. If other available machines have serious system deficiencies, a custom processor design could be a better choice than trying to correct the problem or trying to work around it. Fortunately, as we will see later, there are available machines which are acceptable.

The second advantage of a custom microprogrammed processor comes from microprogramming. This permits efficient I/O device design and flexibility in the macro-level machine because of the very high rate of microinstruction execution. In raw cycle time, however, the microprocessor should eventually win since it eliminates inter-chip signals.

We have selected one of the new commercial microprocessors because it is the correct ultimate solution, and the new crop of components seems to satisfy our requirements in most of the critical areas. As we will see, the choice is not entirely wonderful since it will be necessary to add complexity to the processor node in order to correct for defects in the available microprocessors.

2.1.2 Address Space

Perhaps the most important criterion that we have used to select a processor is the need for a large address space. There are two address spaces involved: the physical address space and the virtual address space. The physical address space must be large enough to hold all of the memory in the machine. The virtual address space must be large enough to hold the entire process: its code and all of its data.

Experience has repeatedly shown that a 16-bit virtual address space is simply too small. Mapping hardware has often been suggested as a mechanism which can be used to expand the virtual address space. While a mapping mechanism is an appropriate mechanism for providing a virtual machine to user processes as in an operating system, its use for expansion of the virtual address space is incorrect. The reason for this is that it is too expensive and too awkward to set the maps prior to every memory reference. As a result, the program must be organized around the use of the maps. This is too large a burden to place on the programmer. As a result, programs end up being squeezed into the true virtual address space (as is the case with the UNIX kernel) or being chopped into many separate processes, with a serious increase in complexity and a serious loss of efficiency.

It is not clear how large these two spaces must be. An argument is made later that a 24-bit physical address space is adequate for the Voice Funnel, but it is very hard to know how large a virtual space is required; at least we know that an address space of 16 bits is too small.

Architecturally, the Z8000 could provide up to a 31-bit byte address. However, only 23 bits of address are currently implemented. The address is partitioned into a seven-bit segment address and a 16-bit segment offset. Thus the Z8000 can directly address up to 8 megabytes of memory. The only drawback of the long address required by the large addressing space is the larger size of the instructions and the need for register pairs for some addressing modes. This problem is minimized in the Z8000 by use of segmented addressing features, by the availability of a large number of general purpose registers, and by the availability of double word data transfers.

2.1.3 Multiplexing Address & Data

The Z8000 multiplexes the low 16-bits of the address (the segment offset portion) and 8 or 16 bits of data on the Address/Data bus. This multiplexing scheme does not decrease throughput, however, since the scheme matches the multiplexing scheme used with current high density random access memory. This same multiplexing scheme will probably also be used on the inter-element bus in the processor node.

2.1.4 Memory Refresh

The Z8000 provides a feature which automatically refreshes dynamic memory. Unfortunately, this feature cannot be used in the processor node because of the potentially long latencies involved in a remote memory access. (These latencies can be on the order of hundreds of microseconds.) During these accesses, the processor would not be able to perform refresh cycles as required for dynamic memory.

As a result, the processor node memory section will employ a separate mechanism for refreshing.

2.1.5 Memory Management Units

Zilog is developing a Memory Management Unit (MMU) which permits implementation of a virtual memory system for the Z8000. Four status lines together with the segment address provide a mechanism for relocation and protection on a segment by segment basis. The separation of processing and memory management has several significant advantages, such as a capability for multiple Memory Management Units, and allows a Direct Memory Access device to access the Memory Management Unit.

2.1.6 Z8000 Interaction with I/O

The Z8000 architecture distinguishes between memory and I/O spaces and thus requires specific I/O instructions. The I/O address is 16 bits wide and access may be made to either bytes or words. This separation allows the use of a short high speed memory bus and a potentially long low speed I/O bus, and it allows simultaneous I/O and processor access to memory.

Both the I/O and memory buses need address, data, and control signals. One design compromise in the Z8000 was to overlay the signals used by the memory and I/O buses. Note that this is an implementation compromise and not an architectural one.

2.1.7 Provisions for Independent DMAs

All I/O device interfaces are connected to the I/O bus which can access the memory thru a Memory Management Unit. Once an I/O device interface becomes I/O bus master, the interface can transfer as many bytes or words as required. In the processor node, I/O device interfaces are allowed to access only local memory, thereby preventing any long latencies that might occur in referencing remote memory. The Z8000 can become I/O bus master simply by executing one of the special I/O instructions. Thus DMAs are independent of the processor but interact with the memory manager.

2.1.8 The Z8000 Interrupt System

The Z8000 provides a vectored interrupt system. Interrupt vectors are used to determine the identity of the device requesting service. Only one interrupt priority level is provided by the Z8000. A control bit in the processor status register provides a means for inhibiting interrupts from all I/O devices.

During the interrupt acknowledge sequence, the interrupting I/O device gates 8 bits of data onto the address/data bus. The Z8000 uses this data as an index into its New Program Status Area to find the new four byte program counter value. Since the first 60 words in the New Program Status Area are reserved for various program traps, only 50 interrupt vectors are allowed. While this limitation at first glance does not appear to be severe (the PDP-11 allows only 56 vectors), it poses serious problems. Many of the LSI I/O devices use interrupt levels freely, causing a shortage of interrupt levels with even a small number of devices.

For example, the Zilog SIO chip uses 8 interrupts. If this chip is employed, only 6 I/O channels can be supported. In order to provide the 32 or so I/O channels that will be connected to a single processor node, we will have to allocate unique interrupt vectors to only those devices which require low latency response. This unfortunately requires some interrupt service routines to determine the interrupt cause for themselves.

2.1.9 Resource Locking

There are certain primitive operations which are specifically important for a multiprocessor, such as a facility for one processor to "lock" a resource so that some other processor cannot also capture it. The locking primitive is often implemented with a read-modify-write memory operation. The Z8000 provides one such instruction, the test and set instruction, which could be used for a lock in local memory since the Z8000 normally holds the bus between the read and the set. However, the Z8000 does not assert any status lines which indicate that it is performing an indivisible operation. As a result, it is difficult for the switch interface to correctly identify this instruction so that a remote test and set instruction could be performed indivisibly.

There are two options. Either the switch interface can watch for the test and clear instruction, or it can provide an independent mechanism such as an explicit exchange operation in the switch interface.

2.2 Processor Extension Peripherals

There are some features which the Z8000 does not provide, such as a real time clock, a communications link for front panel emulation, and a ROM for bootstrap and diagnostics. Fortunately, these functions can be provided by readily available LSI

components. We shall discuss each of the processor extension peripherals in the following sections.

Because of the current unavailability of any Z8000 peripheral devices, we have chosen processor extension peripherals from Zilog's Z80 processor family. While it is not required that we utilize Zilog's peripherals, we will benefit from Zilog's efforts to insure that its Z80A peripheral devices integrate well and work together properly with the Z8000.

2.2.1 Memory Management Unit

The purpose of the memory management unit is to separate the logical address space seen by the processor from the actual memory configuration, the physical address space. The memory manager serves as an interface between these two spaces, translating logical addresses into physical addresses. The time of the translation is also a convenient one to do some ancillary functions, like protection. There are many reasons for providing such a separation but the main idea is to isolate the programmer from the vagaries of the hardware. The program should be insensitive to what processing nodes or memory modules actually exist, or even where a particular piece of data actually resides.

The system that we chose is a simple one. The physical address (as output by the memory manager) is sufficient to locate any word in any processor node in the entire machine. That is,

encoded within it is a processor node number and a location within the memory of that processor node. A process selects which objects it needs to reference and, with the assistance of the operating environment, the local memory manager is set up with the physical addresses of those objects. When an access is made, the memory manager converts the logical address to a physical address and simple address recognition logic decides if that address is located in this processor node. If not, the physical address is handed to the switch interface which tries to get that data by sending out an appropriate transaction on the switch. When this reaches the destination processor node, the incoming request for a word has a physical address contained in the requesting transaction. Thus the switch interface can directly access the local memory for the word and return it.

A complication arises should the operating environment decide to move an object to a different physical address. In that case, all the memory managers which contain a pointer to this object must be updated. The operating environment can do this by informing each processor node each time it wishes to make such a change. If we were planning a classic paging system, with pages of memory moving about all the time, this would not be a good scheme. As it is, we do not envision the need to move things around very much, and so this safe, easy to understand mechanism should not be a burden.

Zilog is planning to introduce a memory management chip to facilitate these functions. It accepts 23-bit virtual addresses: 7 bits of segment and 16 bits of segment offset. The output is a 24-bit physical address.

To form a 24-bit physical address, the 16-bit offset is added to the base for that given segment. Thus the Z8000 can directly address any 8 Mbytes within a 16 Mbyte physical memory. Furthermore, segments may overlap and be of any size that is a multiple of 256 bytes.

In addition to address relocation, the Memory Management Unit also checks a number of segment attributes during each memory reference against the status lines of the Z8000. If a mismatch occurs, a trap is generated. Each Memory Management Unit provides protection for 64 segments. Thus minimaly two MMUs are needed to provide memory protection for the full 128 segments.

2.2.2 Counter/Timer Peripheral

The Z80 Counter Timer Circuit (CTC) has four counter circuits under software control, each of which may be used to cause system interrupts, provide time stamps for vocoder parcels, or generate output signals.

2.2.3 Serial Input/Output Peripheral

The Z80 Serial Input/Output (SIO) peripheral is a dual channel multi-function data communications controller. The device accepts programmed instructions from the Z8000 and supports many serial data communications disciplines, synchronous and asynchronous, full and half duplex.

The SIO is used to allow entry into the system. For example, connecting a terminal to the SIO together with the system debugging code in ROM should enable the operator to investigate the state of any processor, memory location, or peripheral in the system.

2.2.4 Read Only Memory

The Read Only Memory (ROM) consists of two 4K x 8 Erasable PROMs. This ROM is needed to allow execution of instructions immediately upon power on. In addition to the bootstrap, other programs such as the system debugger and some diagnostics would reside in the ROM.

2.2.5 Switch Interface

Before we discuss the switch interface, it is important to distinguish between two methods of using the switch: implicitly and explicitly.

2.2.5.1 Implicit and Explicit Switch Utilization

In implicit switch utilization, accessing a word in local or remote memory simply requires executing an instruction which contains an address which references that word. The memory manager decides whether the word is in local memory. If it is not in local memory, it signals the switch interface that this is a remote memory request. The processor is told to wait (thereby holding up the processor bus), and the switch transmitter assembles an appropriate message and sends it to the destination processor node. When the result returns, the switch interface hands the result to the processor and allows it to continue.

An explicit switch access uses the switch interface like a DMA device. The processor tells the switch interface the starting locations for both the local and remote memory and the size of the block to be transferred. In a local memory to remote memory block transfer, the switch interface assembles the appropriate message by pulling a portion of the block out of local memory and sending it to the destination processor node. When a reply message returns, the switch interface assembles another message containing the next subblock of words out of local memory and so on until the entire block has been sent. In a remote memory to local memory block transfer, the switch interface sends a message to the destination processor node asking it to send a portion of the block in remote memory. The

reply message from the destination processor node contains the block portion requested. The switch interface then loads the data in the reply message into local memory and repeats the process until the entire block is transferred from remote memory.

Implicit switch utilization substantially reduces the overhead to access a remote memory location. For example, consider adding 14 to remote memory location F00. The execution of the instruction ADD #14,F00 and the interaction of processor and switch interface would cause the desired function to be performed. For the same example, explicit utilization would require: 1) a block transfer of F00 to some temporary variable in local memory, 2) adding 14 to that temporary location, and 3) a block transfer of the temporary variable back to F00. Thus while the switch bandwidth utilization in the two cases would be the same, the execution time would be dramatically different.

Although implicit switch utilization has a lower overhead for single word transfers, explicit switch utilization makes better utilization of the switch bandwidth for transfers of more than a couple of words. Explicit switch utilization also allows the processor to execute instructions (at a reduced rate because of local memory contention) while the block transfer is in progress.

2.2.5.2 Switch Interface Functions

The switch interface has four primary functions.

First, the switch interface must respond to a remote processor node's request to access local memory by fetching data from, or loading message data into, local memory and then sending a reply message back to the originating processor node.

Second, the switch interface must detect those references which are remote, send a remote memory access request message, receive the reply, and complete the bus transaction with the processor.

Third, the switch interface must accept block transfer control information, partition the block into appropriate length subblocks, and then send and receive a sequence of messages transferring each subblock.

Fourth, the switch interface must execute several special operations such as interrupting the processor or performing a locking operation.

In addition, the switch interface must retransmit messages that have been rejected either because of switch conflicts or because of contention at the destination. Also, the switch interface must inform the processor when a message cannot be successfully sent to a destination node after a number of tries.

2.2.5.3 Message Classification

There are two groups of messages. The first message type is called a "Request". A request message asks a remote switch interface to do something (e.g., load memory location 1234 with data word 54321). The second message type is called a "Reply". A reply tells the switch interface which has sent a request that the remote switch interface has either positively or negatively responded to the received request.

To prevent a system deadlock, messages in the two groups are treated differently by the switch interface. The switch interface provides two input buffers, one dedicated to receiving replies. There are also two output buffers, one dedicated to transmitting replies.

2.2.5.4 Message Formats

The switch as currently envisioned will move four bits (a nibble) of the message during each switch clock period. It is therefore desirable to create a message format as a concatenation of four-bit pieces so that no bit shifting is required in the switch interface. A message is composed of six segments as shown in Figure 2.2.5.4-1. Some of the message segments may be missing.

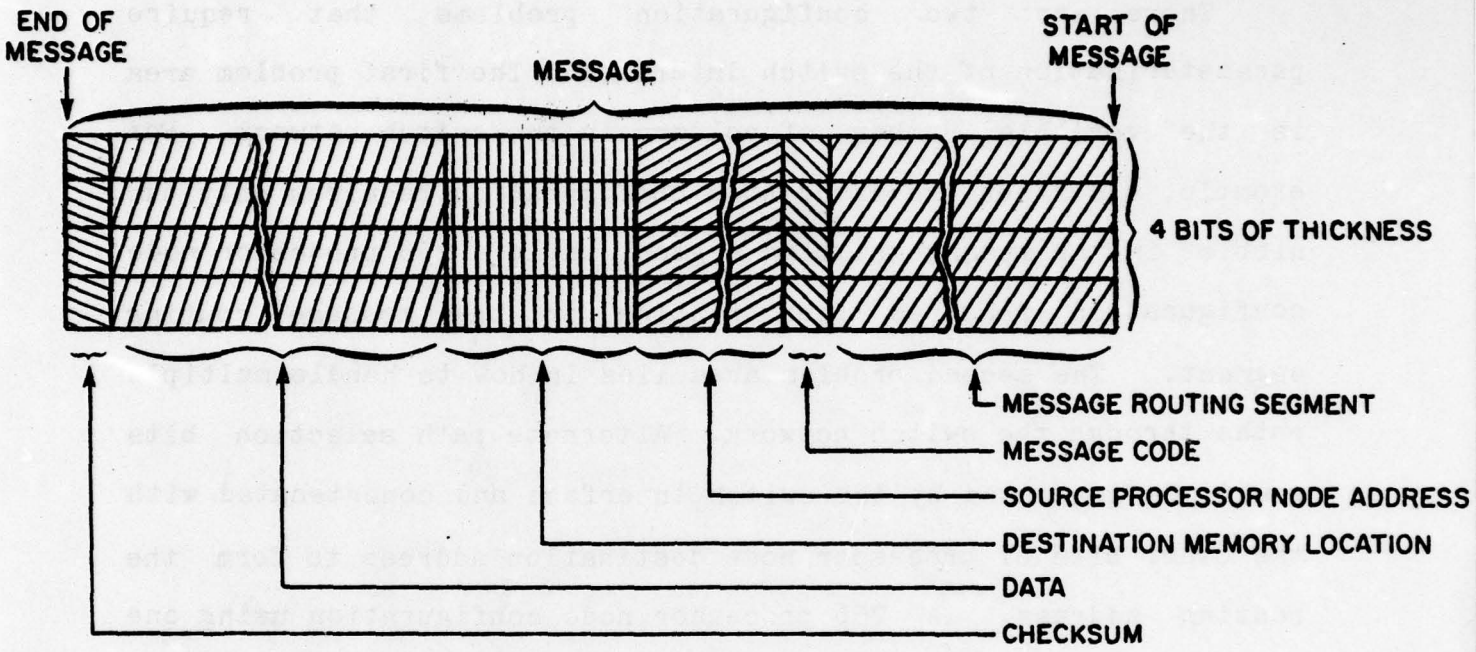


Figure 2.2.5.4-1 Message Format

Each message begins with a routing header that is used by the switch network to route the message to the destination processor node. The number of nibbles in this sequence must therefore equal the number of columns in the switch network. This is a parameter which differs with different switch sizes. Two of the bits in each message routing nibble are used for the routing decision. The other two bits are not used. These routing header bits will be extracted from the destination physical memory address.

There are two configuration problems that require parameterization of the switch interface. The first problem area is the variable number of columns in the switch network. For example, a sixteen processor node configuration requires only two nibbles in the message routing segment while a 256 processor node configuration requires four nibbles in the message routing segment. The second problem area lies in how to handle multiple paths through the switch network. Alternate path selection bits would be generated by the switch interface and concatenated with the other bits of processor node destination address to form the routing address. A 256 processor node configuration using one extra column of switch nodes would provide four paths through the switch and thus require the switch interface to generate two path selection bits. The switch interface must therefore be parameterized to tell it 1) how many nibbles are required in the message routing segment and 2) how many bits of path selection are desired.

The second segment in each message is a four bit message type code. This code is used by the processor node receiving the message to determine what to do with the message contents and how to unpack the message.

The third segment is the source processor node address. It is included in every request and is absent in every reply. It provides the information required by the processor node receiving

a request to generate the routing segment of the reply. Again, the switch interface must be programmable to handle a variable number of bits in the source address. For example, a sixteen processor node system requires only one nibble while a 256 processor node system requires two nibbles.

The fourth segment is a four nibble destination processor node memory address. It is included in every request and is absent in every reply.

The fifth segment consists of a four bit data word count and/or the data contents of the message, depending on the message type.

The last segment is a nibble containing a checksum for the message. The checksum is the sum of all of the nibbles in the message excluding the destination routing nibbles. This provides protection for the data. In order to detect routing errors, the checksum also includes the destination processor node number. This avoids the need to include the destination processor node number in the text of the message. The receiving node computes a checksum on the message it receives and then adds its own processor node number. This checksum technique provides a means of detecting not only data errors, but routing errors as well.

2.3 Memory

The memory in each processor node serves not only as the local memory for the processor and I/O in that processor node, but also as a portion of the global memory space which is accessible from any processor node. As such, each byte in the memory of each processor node must have a unique physical address. Because the memory management IC that is compatible to the Z8000 is limited to a 24-bit byte address, the maximum amount of memory that may be attached to a processor node (assuming that all processor nodes have equal memory) is 2**24/(number of processors). The processor node memory will be set at 128K bytes because 128K bytes is a reasonable memory size using the 16K dynamic memory RAMs, and for a broad spectrum of applications, 128K bytes feels "large enough".

In order to detect memory errors, the memory system will include one parity bit per byte. Byte parity is necessary even though the memory is 16 bits wide because of the provision for byte writes.

2.4 PC Board Element Allocation

The processor node will be implemented on two PC boards, one for the processor, switch transmitter, switch receiver, and memory, and a second for all I/O interfaces. This partitioning permits flexibility in the I/O system without requireing changes

to the processor and switch related components at the same time.

3. Z8000 Performance Evaluation

It has been pointed out that one of the major advantages of the Z8000 is its speed. To investigate this and to gain some insight into the quality of the Z8000 we will investigate one benchmark program. From that benchmark we will get estimates of Z8000 speed and code size. As a point of reference, we will use the PDP-11.

The benchmark consists of a routine that allocates a large local array and calls a routine which returns the sum of that array. This benchmark exhibits calling costs, allocation costs, looping ease, and a reasonable amount of stack manipulation. The BLISS-11 program below is the benchmark:

```
MODULE BenchMark(OPTIMIZE,SAFE,NOUNAMES,FINAL,MAIN)
    = BEGIN

GLOBAL ROUTINE WordArraySum(Array,Size)=
   BEGIN
      WORD LOCAL sum,pointer;
      sum _ 0;
      pointer _ .array;
      DECR counter FROM .Size TO 1
         DO BEGIN
            sum _ .sum + ..pointer;
            pointer _ .pointer + 2;
         END;
      .sum
   END;

GLOBAL ROUTINE Bench=
   BEGIN
      WORD LOCAL vector[300];
      WordArraySum(vector,300);
   END;

END ELUDOM
```

From this benchmark we can hope to get reasonable measures of the costs of the following activities (the abbreviations in parentheses will be used in later tables):

Calling a global routine	(Call)
Returning from a routine	(Ret)
Register allocation	(Sav+Res)
Parameter setup	(Parm Set)
Parameter access	(Parm Acc)
A typical loop iteration	(Loop)
Local space allocation	(Loc Allc)
Local space deallocation	(Loc Dallc)

3.1 PDP-11 Benchmark

The BLISS-11 compiler produced the following code for this benchmark. The code is verbatim except for comments which have been added to the PDP-11 code for clarity.

```
; BLIS11 V.78095          Thursday 21-Sep-78 11:01.26      BENCH.B11
.PDP10
; 0001   MODULE BenchMark(OPTIMIZE,SAFE,NOUNAMES,FINAL,MAIN)
             = BEGIN
; 0002
.SBTTL   GLOBAL ROUTINE WordArraySum(Array,Size)=
; 0003   GLOBAL ROUTINE WordArraySum(Array,Size)=
; 0004      BEGIN
; 0005         WORD LOCAL sum,pointer;
; 0006         sum _ 0;
; 0007         pointer _ .array;
; 0008         DECR counter FROM .Size TO 1
; 0009            DO BEGIN
; 0010               sum _ .sum + ..pointer;
; 0011               pointer _ .pointer + 2;
; 0012            END;
; 0013         .sum
; 0014      END;
; 0015
.TITLE BENCHMARK

.CSECT BENC.C

WORDARRAYSUM:
        JSR     R$1,$SAV2       ; Allocate working space
        CLR     R$0             ; Clear sum
        MOV     12(SP),R$1      ; Get Parms: R1 <- Array pointer
        MOV     10(SP),R$2      ;            R2 <- Array size
        BR      L$4             ; Enter loop, check size zero
L$3:    ADD     (R$1)+,R$0      ; Loop: Add word, advance pntr.
        DEC     R$2             ;       Decrement counter
L$4:    BGT     L$3             ;       Repeat if more entries
        RTS     PC              ; Go home, (Get Regs: coroutine)

; ROUTINE SIZE:  12
```

```
        .GLOBL  WORDARRAYSUM
        .SBTTL  GLOBAL ROUTINE Bench=
        ; 0016  GLOBAL ROUTINE Bench=
        ; 0017     BEGIN
        ; 0018        WORD LOCAL vector[300];
        ; 0019        WordArraySum(vector,300)
        ; 0020     END;
        ; 0021

        .CSECT BENC.C

        BENCH:
                SUB     #1130,SP        ;Allocate: Local array, 300 words
                MOV     #2,-(SP)        ; Set Parms: Array pointer
                ADD     SP,@SP          ;   To array above this on stack
                MOV     #454,-(SP)      ;   Array size is 300 decimal
                JSR     PC,WORDARRAYSUM ; Global routine call
                ADD     #1134,SP        ; Deallc: parms and local array
                RTS     PC              ; Go home

        ; ROUTINE SIZE:  12

        .GLOBL  BENCH
        ; 0022  END ELUDOM


        .GLOBL $SAV2
        .CSECT BENC.G

        $BREG: .BLKW 1

        ; Size:  26+0
        ; Run Time:  0 Seconds
        ; Core Used:  12K
        ; Compilation Complete

        .END BENCHMARK
```

3.2 Z8000 Benchmark

For comparison with the PDP-11 code, here is a hand-coded version of the benchmark for the Z8000. The version shown here is the most efficient way we could find to do the task, compared to, say, the cleanest way to do the task.

```
WORDARRAYSUM:
        PSHL    RR2             ; Allocate: Reg for array pointer
        PSH     R4              ;           Reg for array counter
        CLR     RO              ; Clear the sum we are making
        LDL     RR2,16(SP)      ; Get Parms: Array pointer
        LD      R4,12(SP)       ;            Array size
        JR      CLR,L$2         ; If size is zero: skip loop
L$1:    ADD     RO,(RR2)        ; LOOP: Add next array entry
        INCR    R2,#2           ;       advance the pointer
        DJNZ    R4,L$1          ;       decr cntr and loop?
L$2:    POP     R4              ; Deallocate: Both
        POPL    RR2             ;               Regs
        RET     ALWYS           ; Go home, condition: always

BENCH:
        SUB     SP,#(1130)      ; Allocate: Local word array[300]
        PSHL    (SP),SP         ; Set up Parms: Push and
        INCR    (SP),#4         ;               align array pntr
        PSH     (SP),#454       ;               and array size 300
        CALL    WORDARRAYSUM    ; Call global routine
        ADD     SP,#(1130+6)    ; Deall: Parms and Array
        RET     ALWYS           ; Go Home, condition: Always
```

3.3 A Discussion of the Benchmarks

We will discuss the two routines in the context of the categories mentioned in Section 3.

3.4 Code Size

Both routines are about the same size. The lack of a uniform dual-operand addressing structure for the Z8000 might be expected to cause larger code sizes in some situations. For example, both routines need to do some manipulations of parameters that are on the stack. The PDP-11 version manages this cleanly since memory can be treated identically to registers. The Z8000 manages it by having a convenient

instruction, an increment instruction which is capable of an indirect reference. Since we have a pointer to the parameter, and the increment is small, we can do the address manipulation on the stack.

3.5 Calling a Global Routine

Both versions are effectively equivalent.

3.6 Returning from a Routine

Both versions are fine; the Z8000 has a conditional return that checks the state of the condition codes, an interesting idea.

3.7 Saving and Restoring Registers

The PDP-11 version uses a standard PDP-11 technique for saving registers: a global routine is called that does the save and then does a coroutine return so that when the routine finishes, the registers will be restored automatically. In all cases except the save of one register, on all PDP-11 models, this technique is faster and produces less code. A more traditional technique would look like:

```
MOV     R1,(SP)-        ; Allocate: Register
MOV     R2,(SP)-        ;           and another
<routine body>
MOV     +(SP),R2        ; Deallocate: Register
MOV     +(SP),R1        ;             and another
```

We will assume this is the technique used by the PDP-11 version of the benchmarks. It is interesting, however, to note that coroutine linkages are not a natural programming primitive provided by the Z8000.

The Z8000 register saves are now identical to the PDP-11 technique. The existence of a block register transfer instruction is somewhat irrelevant since it copies upward in memory while the stack grows downward. This means to use it you must first align the stack pointer. In most cases it is easier to use push and pop instructions.

3.8 Local Space Allocation and Deallocation

In both versions this is a simple addition or subtraction from the stack pointer.

3.9 Setting Up Parameters for the Call

The PDP-11 version proceeds by pushing the offset from the stack pointer to the array onto the stack and then adding the stack pointer to it.

The nice quality of this operation is the addition that is done on the stack. This single addition might have cost a register allocation, a move of the stack pointer, an addition, and then the deallocation of the register. The Z8000 version manages the same feat via the increment instruction.

3.10 Making Parameters Accessible

In these examples this category means getting the parameters off the stack and into the registers. Both versions have little trouble with this.

3.11 A Typical Loop Iteration

The loop consists of four parts: addition, pointer advancement, counter decrement, and conditional branch. The PDP-11 version is able to combine the first two, while the Z8000 version is able to combine the second two.

The Z8000 version is more attractive in some ways. The PDP-11 attitude was that the branch would be able to use the last data manipulation result via the condition codes to drive the branching decision. This has not turned out to be true and has often caused practices such as placing a zero value on the end of arrays to act as a flag. The decrement and skip on zero instruction provided by the Z8000 is better; it would be even better, of course, if it had some range of increments, and condition codes.

3.12 Conclusions from the Benchmark

The tables below compare three machines: the LSI-11, the Z8000 running in nonsegmented mode, and the Z8000 running in segmented mode. The LSI-11 is a machine that addresses 2^16

bytes naturally, as is the Z8000 running in nonsegmented mode; the Z8000 running in segmented mode can address 2^23 bytes and hence is slowed somewhat by having to manipulate larger addresses.

The first table below shows code size, memory fetches, and execution times for all three machines. All fetch counts are in 16-bit words, all code sizes are in 16-bit words, and all execution times are in microseconds.

	LSI-11			Nonsegmented Z8000			Segmented Z8000		
Category	Time	Size	Fetch	Time	Size	Fetch	Time	Size	Fetch
Call	6.95	2	2	1.75	2	2	2.5	3	3
Ret	5.25	1	2	1.5	1	2	2.25	1	3
Sav+Res	20.3	4	8	7.0	4	8	8.5	4	10
Parm Set	19.6	5	9	6.0	4	8	6.75	4	9
Parm Acc	12.3	4	6	5.0	4	6	6.5	4	7
Loop	12.6	3	4	3.75	3	4	3.75	3	4
Loc Al	4.9	2	2	1.75	2	2	1.75	2	2
Loc Dall	4.9	2	2	1.75	2	2	1.75	2	2

The next table contains the same values converted to percentages of their LSI-11 equivalents.

Category	Nonsegmented Z8000 Time	Size	Fetches	Segmented Z8000 Time	Size	Fetches
Call	25	100	100	36	150	150
Ret	28	100	100	43	100	150
Sav+Res	34	100	100	42	100	125
Parm Set	31	80	89	34	80	100
Parm Acc	38	100	100	49	100	117
Loop	29	100	100	29	100	100
Loc Allc	36	100	100	36	100	100
Loc Dallc	36	100	100	36	100	100
Averages	32%	98%	98%	38%	104%	118%

In summary:

- The code density of the Z8000 is very similar to that of a PDP-11.
- It is straightforward to code those things that programs do most often.
- A few nonuniformities in the Z8000's instruction set will make code generation more difficult than for the PDP-11, but in general not too much more difficult.
- The execution speed of the Z8000 in the segmented operating mode is 2.6 times that of an LSI-11.
- The average instruction execution time is about 2 microseconds.

Chapter V: System Software

Chapter V: System Software

Contents

1. Introduction

In this chapter we consider various software components which are not specific to the application but which make the application much easier to build, maintain, test, and modify. These components fall into two broad categories: the operating system, which is resident with the application and which supports it at run time; and a collection of utilities which make up the development environment and which are used to maintain and develop the application, the operating system, and one another.

The next several sections of this chapter discuss the operating system, while the last section briefly outlines the principal components of the development environment. The operating system discussion includes an overview of the primary reasons for providing an operating system, a review of the interface presented to the application programmer, and a discussion of the most difficult operating system issues presented by the multiprocessor environment of the Butterfly Multiprocessor: scheduling, memory management, and reliability.

2. Operating System Overview

Experience in a variety of situations has convinced us that we should separate resource management and problem-specific concerns in the Voice Funnel software. Having built systems with and without such a separation, we are painfully aware that when no such separation exists, several significant and costly problems arise. Among the most important of these are that:

- problem-related algorithms become cluttered with details irrelevant to the problem at hand;
- development and maintenance proceeds much more slowly than the intrinsic difficulty of a task would suggest;
- adaptation to changes in strategy is hindered by the inflexibility of the general structure; and
- errors frequently occur which are not directly related to the problem being solved.

With such strong arguments in favor of this separation, it may seem surprising that the alternative is ever considered. There are, of course, dangers, both apparent and real, in creating such a separation. The separation is typically achieved by building an operating system. An operating system is seductive and, if not controlled, it can grow to consume far too much of the personnel, memory, and processor bandwidth available to a project. Also, because it is a separately identified component, an operating system can appear to represent additional or unnecessary work. Of the two concerns, the latter is more apparent than real, since what really occurs is that work which

would otherwise be distributed throughout the software is instead consolidated and shifted from one area into another. The concern about the operating system task growing out of control is real, however, and must be carefully managed.

The approach we propose to take is to build a limited, but potent and extensible, operating system which is sufficient to meet the perceived requirements of the Voice Funnel and which is flexible enough to adapt to changes in those requirements and to the emergence of any future requirements which can reasonably be anticipated. We have begun this task by reviewing those capabilities which have become common in operating systems [BRIN 73] and by examining any special requirements which are imposed by the Voice Funnel task or the Butterfly Multiprocessor hardware. We have attempted to select those capabilities needed in one form or another for the Voice Funnel and to design an operating system which provides those capabilities while showing promise of being extended to meet additional requirements, should they arise.

Throughout the design of both the Voice Funnel and the operating system software, we have paid considerable attention to the choice of services to be provided by the operating system, for it is not the number or complexity of the operating system features which matters, but rather their simplicity, power, and cost. Time spent in reducing the overall conception to a small

set of elementary, powerful, and inexpensive facilities will pay off handsomely in reduced time to develop both the application and the operating system, in reduced consumption of processor cycles, and in increased effective utilization of personnel and hardware.

Three concepts in particular have guided us in pursuing these objectives. We have tried to be conscious of what the programmer can do himself as easily as can be done automatically. We have tried to avoid building attractive facilities for which we cannot see a clear justification in the current application. Finally, we have taken advantage of the fact that the Voice Funnel is a dedicated application. This will allow us to avoid much of the checking and enforcement which would be required in an environment which had to run arbitrary user programs. Instead, we will be able to limit checking to those conditions which might be expected to arise from routine hardware or software failures rather than from malicious behavior or outright negligence.

The principal facilities of the proposed operating system are briefly described in the following paragraphs. The major areas, processor management (or scheduling), memory management, and reliability, will be discussed in depth in subsequent sections.

- Processor Management (or Scheduling): Allows a large number of processes* to share one or more processors without requiring any of them to know about multiple processors, about which processor they might run on, or about what to do with time they are not using. Assigns work to processors, shares available processor time based on a mix of priority and fairness considerations, and attempts to maintain more or less level load on the various processors. Permits processes to run concurrently (i.e., overlap in time), and permits them to operate asynchronously (by buffering or queuing work or data passing between them).

- Memory Management: Ensures that each process has transparent access to the memory assigned to it, allows protection of the private memory (e.g., instructions and stack) of any process from any other process, and allows voluntary sharing of memory by cooperating processes. Relieves each process of the task of explicitly managing the memory mapping registers, and protects each from incorrect use of the registers by the others.

- Inter-process Communication: Facilitates the transfer of messages or data from one process to another, either through shared memory or by copying data from one address space to the other. Provides simple send and receive mechanisms which relieve the user of the need to perform mutual exclusion, memory management, and scheduling operations directly.

- Process Synchronization: Provides for coordinated use of shared data, preventing conflicting access or modification by multiple processes. Provides for coordination between processes awaiting or causing common events.

- Interrupt Handling: Provides a common mechanism for servicing interrupts and for utilizing them to keep work flowing briskly through the system.

- Reliability and Availability: Provides mechanisms for (1) periodically assessing the health of all hardware, all

* We will use process in its customary computer science sense to refer to the various independent (and perhaps cooperating) software components (or programs) which may be given control of a processor by a scheduler. We will use task (which some computer scientists have used in place of process) in its everyday sense to mean a piece of work to be done.

software and all critical data bases; (2) identifying and removing from use any failed components; and (3) realigning the remaining components to continue operation at reduced capacity (graceful degradation). Presents a reliable framework within which the application can run without itself having to attend to all of the details.

A framework such as we have described presents a number of important advantages. The Voice Funnel software will be organized by function, with capabilities that are required at many points being concentrated into a small number of elementary and powerful functions which can be built and tested once and then used repeatedly. Problems intrinsic to the application will be clearly separated from problems of taming the environment, with the result that solutions of both types can be developed and expressed independently. This will yield simpler and more effective solutions in both areas. It will make it easier to experiment with new algorithms or to incorporate new requirements, since the number of problems requiring simultaneous solutions will be reduced. It will also increase the safety of critical data, since the data will be handled in fewer places. While there will certainly be some cases in which this solution will be less efficient than one in which the application handles all resource problems in line, careful attention to the potential dangers of this approach will enable us to minimize them. The gains realized through more effective global control and optimization, through greater flexibility and adaptability, and

through greater reliability, will yield benefits much greater than the occasional inefficiencies incurred.

3. Programmer's View of the Environment

The operating system provides a user environment which attempts to insulate the application software from the raw hardware, while retaining access to the capabilities provided by the hardware and not imposing much overhead. The operating system attempts to support only the application; it is not a general purpose facility and does not attempt to support program development. Program development tools, such as compilers and assemblers, are assumed to be available elsewhere.

To support the Voice Funnel application, we have attempted to make the operating system impose as little overhead or delay as possible upon the processing of individual voice parcels. Where time-critical application modules cannot afford the cost of elaborate operating system facilities, we have endeavored to provide services which can be used without significant overhead or delay.

3.1 Exploiting Parallelism

Numerous workers have attempted to achieve parallelism at various levels in the program decomposition hierarchy [BAER 73]. This has been attempted at the following levels, among others:

- at the program level (which has occurred most often),
- at the procedure level (for example, PL/I for the IBM 370),

- at the statement level (for example, the parbegin/parend construction of Dijkstra [DIJK 65], or the ALGOL 68 compiler for CMU's C.mmp multiprocessor [KNUE 76]),
- at the sub-expression level (primarily in theoretical work), and
- at the instruction level (for example, the CDC 6600 [THOR 64]).

Of course, at each level, there may be sequences of objects (e.g., a group of statements which must be executed sequentially) for which ordering must be preserved.

For various reasons we have chosen to focus at the program level in providing parallelism within the Voice Funnel. The application contains enough parallelism at this level to take effective advantage of our multiprocessor architecture. In addition, this approach provides good control over locality of memory references, which greatly influences the level of hardware efficiency achieved. Finally, by using this approach, we can use an existing compiler (with a new Z8000 code generator), rather than having to write a new compiler or undertake major structural changes to an existing one.

Adopting the second approach (procedure level parallelism) would require modifying the compiler to use some mechanism other than a stack for automatic variables, subroutine linkage, and temporary storage, since multiple processors could not share a stack. In addition, whenever multiple processors were applied to procedures within a program, any data inherited from the calling

procedure, and perhaps the instructions, would be remote to the new processors. For the Butterfly Multiprocessor, this might have significant adverse effects on execution efficiency if the ratio of remote to local references became too large. Finally, the task of writing programs to utilize this degree of parallelism would be more difficult.

The third approach (statement level parallelism) would have all of the difficulties associated with the procedure level approach and would place additional burdens on the programmer and the compiler. At this level, there are two principal sub-approaches available: having the compiler recognize and exploit opportunities for parallelism, and having the programmer do it [DIJK 65]. The former sub-approach obviously requires both a clever compiler and a clever run-time system [KNUE 76]. Both approaches, however, require that storage management similar to that required for parallel procedures be employed wherever the compiler or the programmer introduces parallelism (i.e., much more often then would otherwise occur). Programming at this level would be much more difficult than at earlier levels because the programmer would have to write true multiprocessor programs, rather than cooperating uniprocessor programs or procedures.

Work on the fourth approach (sub-expression parallelism) has been largely theoretical [BAER 73] and is not of interest here. Obviously, it requires a sophisticated compiler, and it may well

require special processor capabilities. Because the burden of detecting and exploiting parallelism would be shifted from the programmer to the compiler and/or the processor, the programming task would be easier than for the previous approach.

The fifth approach is not really a multiprocessor approach. Instead, it has been applied to large uniprocessors with multiple function units capable of executing several instructions simultaneously. Like the fourth approach, it is not of real interest here.

A second major choice, in addition to selecting the level at which to apply parallelism, concerns control mechanisms to be employed for switching control between parallel streams. In particular, it concerns whether the system scheduler should be involved in all decisions to switch control, or whether some control switching decisions might be made more efficiently by closely cooperating streams within an application. The former approach has usually been taken, and in such cases the parallel streams have normally been called processes. More recently, certain workers [KNUE 76] have suggested another level of scheduling in which each process might be allocated one or more processors which it would then switch amongst various internal activities using much simpler mechanisms than a scheduler might use.

We consider the concept of scheduling activities to be a very intriguing area for further study. However, we will not use it for the Voice Funnel because it seems most appropriate when parallelism occurs at the statement level or the procedure level and when the frequency of non-local references does not strongly affect system efficiency.

A quite different attempt at eliminating the scheduler was used in the Pluribus Multiprocessor [KATS 78]. In the Pluribus, programs are written as strips. Strips have two very important properties: they must complete within a very limited time (determined by device latency requirements), and they may not preserve any private context when they finish (since they have no context, context switching is avoided). Work to be done by the strips consists of either I/O device service or work that has been queued internally. Each device and each queue has a priority. Extremely fast dispatching is provided by a hardware unit from which a processor which has completed a strip can determine the identity of the highest priority device or queue requiring service. Despite the advantages of rapid dispatching and minimal context switching, the Pluribus approach has proven less flexible and more difficult to program than the other approaches considered. In fact, recent work on the Pluribus has involved superimposing a process mechanism upon the strip mechanism.

Much of the foregoing has been independent of the specific characteristics of the Voice Funnel. It would, therefore, be appropriate to consider why the choices made thus far are sensible for that application.

The Voice Funnel will comprise a large number of independent data streams going to or from various encoders. Each data stream will comprise a large number of items (control requests and voice parcels). Each item will pass through a number of processing stages (encoder input, one or more multiplexing stages, output to the PSAT, and the reverse), both as an individual item and as a member of an aggregate. While end-to-end sequencing within data streams must be observed, sequencing between data streams need not be.

The work to be done by the Voice Funnel thus breaks down naturally into a number of tasks corresponding to the processing of a single item or aggregate of items at a particular stage. Except that any item must move through its stages in sequence, and except that items from the same data stream must be delivered in sequence at their destination, it will generally be possible to perform tasks in parallel. Moreover, because processing stages will typically be very small, further decomposition of tasks will not usually be attractive because even negligible overhead would become large as task size became very small.

Given this breakdown of the work, it is natural to view the application software as comprising a number of programs each of which will be able to perform one of the tasks which make up the application. Because we have initially chosen to allow parallelism only at the program level, and because we do not intend to allow processes to perform sub-scheduling (i.e., we will not utilize activities), each program servicing a task will be a process.

Two further points deserve brief mention in establishing this view of program organization: the deployment of processes to achieve parallelism and the circumstances under which processes will be created or removed. These will be discussed in the context of the Voice Funnel application, and then some more general comments will be made.

In order to achieve parallelism between data streams, it will be necessary to process several tasks of the same type simultaneously on a number of processors. This will require several instances of each process for which parallelism is desired, since only one processor at a time can run a single instance of a process.* While all such instances might share a

* Where it is necessary to distinguish between them, we will refer to the individual instances of a process as instances or as specific processes, and we will refer to the process in general as the generic process. Similarly, we will refer to individual occurrences of a task as occurrences or as specific tasks and to the task in general as a generic task. Recall that processes are instruction streams and that tasks are units of work.

copy of the code, processor utilization will be much more efficient if code is always local to the processor on which it runs. Therefore, each processor which will run a process will have a copy of the code. Since each instance of a process must have its own stack and private data, these will also be replicated, and for efficiency reasons will also be local to the processor on which they will run. By making multiple copies of all processes, whether or not their instances will actually run in parallel, we will provide the scheduler with more choices as to where to run each process, and we will provide redundancy as protection against the loss of any single processor.

Most Voice Funnel processes will be created when the application starts and will cycle indefinitely, processing tasks successively as they arrive for service. In addition, it will be possible to create or remove processes at any time it is desirable to do so. We have chosen to create processes in anticipation of need, rather than creating processes on demand (as would be done for a time-sharing system), because we know in advance which tasks will occur and that they will recur at frequent intervals. We also do this because the Voice Funnel cannot afford either the delay from loading code or the delay from allocating a stack and private data and updating the scheduling tables each time a task arrives for service. Because the most commonly occurring tasks will be very brief and will be

performed without any intermediate waits, there will be no advantage to interleaving the execution of two instances of one process on the same processor. Thus, we do not anticipate normally having more than one instance of a process on any processor.

Because the Voice Funnel will be a dedicated application, we have endeavored to avoid requiring that the operating system contain facilities which would add unnecessary overhead or delay to Voice Funnel operation. On the other hand, we have attempted to define operating system facilities which can be provided in a manner which will permit their use in other situations. Thus, while we anticipate that Voice Funnel processes will not have to share the system, alternative applications (such as a time-sharing system or another dedicated application) could use the same facilities (perhaps with elaboration) to accomplish different ends. Thus, if it suited a particular application, processes might terminate voluntarily rather than cycling, multiple copies might not be loaded, and processes might be loaded only when needed.

The operating system will be distributed throughout the machine. In order to present all processes in every processor with a compatible environment, the interface will be kept identical on every processor. Internally, some of the operating system may not be replicated on all processors in order to

conserve main storage when little or no performance penalty would result. However, by maintaining a consistent interface, we will be able to write the application code for the Voice Funnel without concern for this.

3.2 Memory Protection

Two mechanisms exist which provide memory protection facilities to the programmer: the compiler (we will be using the C Programming Language; see Section 7.2) and the memory management hardware. The former will provide a softer form of protection, in that it will only detect errors of form, while the latter will detect and prevent all illegal references to protected areas. Since the Voice Funnel will be a dedicated application, these features will not be as necessary as they would be in an open system. However, they will be of immense help in detecting errors during testing, and the memory management unit will detect and contain many unanticipated problems which arise during operation.

The primary advantage of the compiler will be type checking. Although the compiler cannot ensure that values are correct, it will ensure that variables are used validly (for example, that pointers to objects of a particular type are only assigned values which are pointers to objects of that type and that they are only used to reference objects of that type). In addition, use of a

compiler will facilitate the practice of hiding data by declaring variables only in subroutines which may legitimately access them. Both type checking and data hiding will significantly reduce the likelihood of invalid references.

The hardware offers facilities to isolate processes from other processes and from data to which they do not need access. Although not strictly essential, these facilities will prevent processes from making illegal transfer of control to each other and from illegally accessing the private data of other processes. This will prevent possible damage to the system or to other processes when unanticipated hardware or software faults occur and will permit many errors to be recognized more quickly. To provide this support, the Z8000 hardware will allow us to divide the memory seen by each process into as many as 128 segments of various lengths. Each of these segments may be assigned one of several combinations of protection attributes, and each segment may be shared with one or more other processes or concealed from them.

There is a cost associated with this protection which will be discussed in some detail in the section on memory management. To minimize protection overhead, we will group related segments together into domains, which will consist of segments that are to be treated as a unit for sharing purposes. The linker, the loader, and the run-time context switcher will all be aware of

domains and will be able to treat them as units without always having to consider their members individually. By making good use of domains, the programmer can have hardware-assisted memory protection without excessive overhead.

3.3 Communication

The user will see a fairly rich environment for inter-process communication. Facilities will include shared memory, pipes (a variant of the concept described in [RITC 78]), and signals (which are described in the next section). With these facilities, the user may choose among generality, ease of use, and speed, as appropriate. The availability of these communication primitives (including the more basic synchronization primitives) will allow us to select the techniques most suitable for each situation.

The availability of shared memory will provide a facility for communicating without operating system overhead. In such cases, it will be the responsibility of the communicating processes to synchronize their accesses (see the next section).

Pipes will provide a more sophisticated (though more expensive) communication mechanism. Once a pipe is established between two processes, one process (the sender) may send a message to the other (the receiver). Pipes may be initialized to return immediately after attempting an operation or to return

only after the operation has been completed. In the former case, status will be returned to indicate whether or not the operation was completed successfully. In the latter, the calling process will be blocked* until it has been completed. While somewhat more difficult to use, pipes which return status, rather than blocking, provide better control for cases in which it is necessary for a process to have more than one input pipe (see [HAVE 78]). Depending upon the value of flags set when the pipe was initialized, the message itself, or just a pointer, may be moved between the processes.

More complex pipes will also be available. In addition to being set up with one sender and one receiver, pipes may be initialized to have many senders or many receivers (or both). This will permit multiple copies of a process to use a common pipe, hence remaining indistinguishable to processes at the other end of the pipe.

3.4 Synchronization

There will be several facilities for synchronization (or signaling) among processes and for synchronization between a process and a device service routine. These facilities will

---

* When a process is blocked, control of the processor will be returned to the scheduler for allocation to another process. When the condition for which the process was blocked has been met, the process will again become eligible to receive control.

allow a process to wait, either by blocking or by busy waiting (looping), until some synchronizing condition is met. Given these facilities, a process will be able to handle real-time events with relative ease.

Spin locks will be provided for those situations in which a process must have exclusive access to a resource, but will only hold it for a very short time. Any process wanting access to a resource protected by a spin lock must first seize the lock. When a process attempts to seize a spin lock, the process will be placed in a busy wait (repeatedly attempting to seize the lock) if another process already has the lock. As soon as the lock is released, the next waiting process to reference it will gain control (any other waiting processes will continue to wait). Note that the use of a busy wait, rather than blocking, will be acceptable only if the resource will never be held for a long time and its use will not be so frequent that it will be possible for some processor to be locked out indefinitely.

For locks involving waits of longer duration, semaphores [DIJK 68] will be provided. Semaphores will differ from spin locks in that blocking will be used in place of busy waiting, so that other processes may be run while the blocked process awaits release of the lock. Both spin locks and semaphores will be quite primitive, and they will use the same notation. A lock will be initialized as either a spin lock or a semaphore, and the

lock service routine will execute the form of locking which has been specified. In addition to being used for locking shared resources, semaphores may be used for a variety of more general synchronization purposes, such as indicating that some condition has been met (for example, the inter-process communication facility might use semaphores to indicate the availability of a message in a buffer).

We will also provide eventcounts and sequencers [REED 78], which can be used for problems similar to those for which one would use semaphores, but which often yield simpler solutions. Eventcounts will be used to count the occurrences of a particular user- or system-defined event. Processes may read, advance, or wait on eventcounts. Sequencers may be used to order events. They are like the ticket machines in bakeries: processes about to wait on an eventcount can obtain a ticket and wait for the eventcount to be advanced to the value indicated on the ticket, thus assuring that they will be served in turn. A significant advantage of eventcounts and sequencers is that solutions which use them typically involve much less write contention than semaphore solutions, because very often processes will be either readers or writers but not both. This feature may allow us to significantly reduce cross-switch contention for synchronizing variables.

Although semaphores and eventcounts/sequencers provide similar capabilities, we plan to implement both. We will provide semaphores because they have been used for some time and appear in many published algorithms which we might want to exploit directly. However, while eventcounts and sequencers are relatively new, they show promise of providing more straightforward solutions to a number of problems, and we are interested in exploring them further.

3.5 High Bandwidth Considerations

The bandwidth and latency requirements of the application have provided a strong motivation for much of the operating system design. For example, when we provide a facility which is easy to use, but perhaps slow (e.g., the form of pipe for which the system moves the data from one buffer to another), we will also attempt to provide a faster mechanism requiring slightly more effort on the part of the programmer (e.g., the form of pipe which merely transfers a pointer, but requires the processes using it to have a shared data area).

We have also designed most operating system facilities so that run time effort will be front-loaded. Thus, as much information as possible will be specified when a facility is set up for use by a process. The intent is that the "main line" of execution be kept minimal by making as many decisions as possible

when a facility is initialized, so that actual usage of the facility will be streamlined.

3.6 Notes on Local and Remote Memory References

Local memory references are memory references which do not use the switch because they are made to memory which is local to a processor. Remote memory references are references to memory which is local to another processor. These references must use the switch and will take much longer than local references. Occasionally, remote will be used as a generic term for references which cannot be guaranteed to be local.

Recent studies [BELL 78] have indicated that most memory references are to code, to the stack, or to local variables (which the C compiler allocates on the stack), and that relatively few references (typically between 2% and 10%) are to shared data. Because the code, stack, and private data for a process will always be local to the processor which executes the process (see Section 4.1), the number of remote references will generally be small. Because we expect remote references to be relatively infrequent, it will not normally be necessary for the application programmer to worry about where (i.e., in which processor) data may be located, which should simplify programming significantly. There are, however, three cases in which it may be desirable to reflect an awareness of data locality.

When several identical servers exist for a particular task and it is expected that certain shared data will be referenced extensively, the application programmer may indicate that it would be preferable for the task to be handled by a server located on the same processor as the data. This preference will be expressed as a parameter when one process sends a task to another for service.

A second circumstance which might require that locality preferences be specified would arise if the programmer wished two data structures to be local to one another. In this case, he would express the preference when allocating space for the data (at link time or at run time).

Finally, the applications programmer might wish that two (or more) data structures not reside in the same processor's local memory. This might be to reduce contention or to provide safe, redundant copies of a data structure.

4. Scheduler

Processor time in the Voice Funnel will be allocated by a combination of hardware and software. Hardware will allocate time to interrupt service routines when interrupts occur. Time not spent servicing interrupts will be left at the disposal of a software module called the scheduler, which will allocate the time among the various processes in the system (we will use "process" to refer to any instruction stream to which the scheduler may allocate time). Processes may indicate that they will be ready to run only when certain conditions have been met, enabling them to synchronize with one another and with their input sources and output destinations. The scheduler will be free to run ready processes in any order or in parallel, and to interrupt one process in order to run another.

In an ideal multiprocessor, the availability of several processors would require only a straightforward generalization of single processor scheduling techniques: when invoked, the scheduler would apply its scheduling policy to select the most eligible processes and assign the available processors to them. Because all processors would be interchangeable (in the ideal case), the scheduler would only need to assign eligible processes as processors became available, and would not need to consider whether processes and processors were compatible. In practice, unfortunately, real multiprocessors (especially those with many processors) are not ideal, for reasons such as the following:

- connectivity: in some multiprocessor systems, each processor may find some parts of memory harder to reach than others (e.g., special registers may have to be loaded); in some cases, a processor may even be unable to reach certain parts of memory directly, and may require assistance from another processor to do so;

- memory access delay: the various areas of memory accessible to a processor may have different access times, and processes which make many references to slower memory may run much more slowly (or they may run more slowly on processors from which the memory they wish to access has longer access times);

- contention: when several processors share a common memory, or a common path to memory, some processors will have to wait for others when more processors than can be supported simultaneously try to use a path or reference a memory unit at one time; if contention is very high, it may be necessary to allocate memory in a way which reduces conflicts;

- homogeneity: in some multiprocessor systems, not all processors are identical, and a particular process may not be able to run at all on certain processors.

With respect to these considerations, the Butterfly Multiprocessor can be characterized as follows:

- Connectivity will be uniform on the Butterfly Multiprocessor; all memory will be available to all processors and will be accessible in a uniform manner;

- Access times will not be uniform; as described in Chapter IV, memory will be divided equally among processors, and each processor and its local memory will be packaged together; for efficiency reasons, each processor will bypass the switch when accessing its local memory; thus, access times for local memory references will be comparable to normal access times for a uniprocessor, while remote references will be slower due to switch delay;

- Our simulations (see the switch performance graphs in Chapter III) show that if the ratio of remote to local references is too high, contention will be a problem on the Butterfly Multiprocessor, as it would be on any large

multiprocessor which it would be practical to build with today's technology; efforts must therefore be made to reduce the frequency of non-local references;

- Finally, the Butterfly Multiprocessor hardware will be effectively homogeneous; nevertheless, certain differences in the hardware configuration (such as having direct access to the PSAT from only one or two nodes), as well as certain characteristics of the application (such as the need to concentrate processor bandwidth at certain points), may make it desirable to introduce some specialization of nodes through software.

Taken together, these considerations may be referred to as locality considerations. Collectively, they are very significant factors affecting the overall efficiency which can be achieved with the Butterfly Multiprocessor; and they will affect scheduling, memory management, and loading strategies, as well as the design strategy for the Voice Funnel application. In particular, all strategies must be evaluated in terms of locality considerations, although the operating system will endeavor to arrange things so that the impact of locality on the application will be minimized or eliminated, just as the switch will eliminate connectivity as an issue for the operating system.

As stated in Section 3.1, the work to be done by the Voice Funnel will be divided into a number of generic tasks,* which can be performed independently and in any order. While most tasks will be application tasks, some system services may also be

* Recall that processes are units of software to which processor time will be allocated by the scheduler, while tasks are units of work which are performed by processes.

performed as tasks, rather than being performed synchronously. In addition, there may be certain system tasks which are performed periodically or in background (e.g., housekeeping or reliability tasks), which do not arise directly from application requests. Corresponding to each generic task there will be a generic process capable of performing the task. In order to be able to perform occurrences of the same generic task in parallel, there will be multiple instances of each generic process.

In contrast to what might be done for an ideal multiprocessor, we will perform scheduling at two levels. Global scheduling will assign tasks to processors, making decisions as necessary to reflect locality considerations. Local scheduling will allocate processor time among the processes assigned to a processor, using more or less standard uniprocessor scheduling techniques (see, for example, [BRIN 73]).

Global scheduling may be thought of as associating (or binding) tasks, processes, and processors to one another in order to get work done. If we assume that tasks and processors will be bound indirectly, by virtue of being bound to the same process, the interesting bindings to consider will be between tasks and processes and between processes and processors.

Considering the latter case first, we note that a process may be bound to a processor at any of three points:

- when the scheduler decides to run the process,
- when the process is bound to a task, or
- when the process is created.

Binding a process to a processor each time the scheduler decides to run the process would be the most flexible approach (and is the one found most often in textbooks). However, making availability of processors a more important criterion than locality would not be suitable for the Butterfly Multiprocessor (as will be shown in Section 4.1), or for any other multiprocessor which could be built today to support very large numbers of processors.

Binding a process to a processor when scheduling a task for service would be appropriate in many cases (for example, time-sharing systems). However, it would entail a delay (allocating a stack and a private data area, updating scheduling tables, and perhaps loading code) which would be intolerable for most Voice Funnel tasks.

Binding a process to a processor when the process is created will allow us to consider locality, since the instructions, the stack, and the private data for the process can be made local to the processor. In addition, creating processes in anticipation of need (since we know which tasks will occur), and keeping them alive indefinitely (since we know that tasks will recur), will

allow us to avoid unnecessary delay. In a similar manner, the Pluribus [KATS 78] binds strips to processors at system load time.

The problem which remains concerns when to bind tasks to processes and by what rule. Because this decision represents our only real opportunity to balance load on the system and to control variations in the delay encountered by specific tasks, we will let the question of when to make the assignment be driven by the scheduling algorithm chosen.

A number of global scheduling algorithms have been and are being considered, of which three will be discussed in later parts of this section, together with proposed criteria for choosing among them. These three algorithms are:

- an implicit algorithm having no explicit global scheduling module, but instead relying upon natural tendencies in the system to cause tasks to flow toward processors which are less busy; it would be the simplest of the algorithms and would have the least overhead;
- a decentralized, but explicit, global scheduler which would pull tasks toward processors as capacity became available to them; and
- a centralized, but migrating, global scheduler which would push tasks towards the processors with the most excess capacity.

4.1 Analysis of Locality as it Affects Scheduling

The key to effective scheduling on a machine like the Butterfly Multiprocessor is to recognize at any point in time where one is in the spectrum from loosely coupled to tightly coupled. In this context, a tightly coupled multiprocessor is one for which all processors (from the set of homogeneous processors on which a process might be run) are interchangeable for purposes of running a particular process, while a loosely coupled multiprocessor is one for which a particular processor may be much better than any other for running a particular process. In addition to these two extremes, there is a broad range of intermediate cases.

For our purposes, the concept of tightly vs. loosely coupled is equivalent to one of locality or preference. When a system is tightly coupled, there will be no preference as to which processor (in a class) handles each process, while in a loosely coupled system there will always be a strong preference. Locality is relevant, because whether or not code or data are local to the processor on which a process will run will determine the efficiency of running that process on that processor. Preference will depend on efficiency, and efficiency will be a function of locality, reference frequency, and memory and switch contention.

To assess these factors, it is useful to classify memory references as follows:

INSTRUCTIONS

PRIVATE DATA
- static
- dynamic

STACK DATA
- private automatic data
- function linkage data

SHARED DATA
- static
- dynamic*

While we do not yet have measurements of reference frequency for the Voice Funnel, we can make some reasonable estimates based on our knowledge of the Z8000 and our experience with programs of the type we will be writing. Because every instruction executed must be fetched from memory, 50% is certainly a lower bound on the frequency of instruction references. Moreover, since Z8000 instructions which access memory require up to five words, and since many instructions get their data from registers rather than

---

* Private data are internal to one process and invisible to others. Shared data are those which are visible to more than one process. Static variables have unique addresses determined at link time for each process and exist for the lifetime of a process. Dynamic variables have their addresses set at run time by a free space allocator and exist until the space they occupy is explicitly freed. Stack data are private data which are kept on the stack, rather than assigned permanent storage or allocated from free space. Automatic variables are allocated on the stack when the function in which they are defined is invoked, and they cease to exist when the function returns to the function which called it.

memory, 75-85% is a more likely figure. Since automatic variables and function linkage data will be kept on the stack, and since automatic variables are typically the most frequently referenced variables, stack references should account for most of the remaining references (10-20%). References to private variables might account for up to 5% more, leaving perhaps 5-10% for references to shared data. In tabular form this would be:

INSTRUCTIONS	75 - 85%
PRIVATE DATA	0 - 5%
STACK	10 - 20%
SHARED DATA	5 - 10%

Measurements made at Carnegie Mellon University on their C.m* system have yielded similar results [BELL 78]. Measurements made on the Pluribus Multiprocessor indicate that 25% of its references are global, but this is atypical because Pluribus strips have no stacks and keep even their private data in common memory; also some instructions are executed from common memory.

Since remote references will be much slower than local ones, it would be prohibitively expensive to run a process on a processor for which the instructions were not local. To a lesser extent, it will be very desirable to place both stack and private data local to the process which will use them. Shared variables, by their nature, cannot normally be made local to all processes

which might want to access them. However, because they are not referenced too frequently, their effect on overall efficiency will not be too serious.

If one assumes that instructions, private data, and the stack will always be local, that (worst case) shared data will always be remote, and that the effect of remote references will be as shown in the switch performance graphs in Chapter III, the following estimates for processor efficiency (defined in Chapter III, Section 8.1) can be made:

	Processor Efficiency (%)		
	% of Remote References		
	2	5	10
16 Processors	96	89	75
64 Processors	92	78	58
256 Processors	78	52	30

Based on the foregoing analysis, which will be reviewed as we build and measure the Voice Funnel, we can see that, with respect to the specific instances of a process, the Butterfly Multiprocessor is loosely coupled. For any instance of a process we would strongly prefer that the instructions, stack, and private data be located together and that the process be scheduled to run on the processor in whose memory its constituents are located. We will, therefore, load each instance of a process so that its instructions, stack, and data are in the

same memory, and we will only schedule a specific instance to run on the processor in whose memory it has been loaded.

Note, however, that unless the frequency of remote references approaches 10%, or the number of processors becomes very large, we can ignore the location of shared data and still operate with acceptable efficiency. This will greatly simplify both the application and the operating system, which will increase efficiency and make it much easier to exploit parallelism. More significantly, however, it will make it possible to schedule a generic process on any available processor on which an instance of the process has been loaded.

Thus, to the extent that we are prepared to ignore the location of shared data, and to the extent that a generic process is replicated, there are points at which the Butterfly Multiprocessor can be considered to be tightly coupled. In fact, the point at which the scheduler associates a new task with a particular instance of a process is the instant of maximum coupling. Once processing of a task has begun, information will begin to build up in the stack and private area of the specific process chosen, and no other process or processor may be chosen to continue processing the task without loss of efficiency.

4.2 Global Scheduler

As indicated in the introduction to Section 4, there will be two levels of scheduling: global and local. The global scheduler will assign tasks to processes and hence to processors, while the local scheduler will concern itself only with determining which of the processes assigned to its processor should be run next.

At the global level there will in fact be two types of scheduling: initial scheduling and re-scheduling. Initial scheduling will consist of assigning tasks to processors, and will be discussed in detail. When it can be done at all, re-scheduling will be done only when a task cannot be completed by the process to which it was originally assigned (for example, because of overload or failure). Rescheduling may merely require starting a task over on another processor, or it may involve copying the private data and stack of a specific process, or even the instructions. Re-scheduling will not occur often and will probably only be performed for low priority processes. Because re-scheduling will be straightforward (even if awkward), it will not be considered extensively in this report.

Although the processors in the Butterfly Multiprocessor will be essentially homogeneous, some important differences will exist. These differences will arise because specific devices (e.g., particular vocoders) will be accessible only through the

processor to which they are attached, while certain generic devices, such as PSAT interfaces, will only be available on some processors. These hardware differences may lead to some specialization by individual processors or groups of processors (although we will attempt to minimize this). Further specialization may be induced by the algorithm adopted for the application (for example, it may be necessary to concentrate bandwidth at some points, while having other points which cater to a large number of low bandwidth activities). Because of either hardware differences or software specialization, some processes may be loaded only on selected nodes. As a result, the global scheduler will have to consider both the availability of the required service and the current workload of a node before sending a task there for service.

Depending on the specific algorithm chosen, the global scheduler may or may not attempt to assign a task to a process on a particular processor. If the global scheduler is willing to consider any process, however, it will be doing so at the time at which the system is most tightly coupled (with respect to the decision to be made). Once the assignment of the task to a process has been made, it will not be reconsidered, except under unusual circumstances, because once processing has begun the specific process chosen will have developed local context which would be difficult to move.

At system start-up, several instances of each process will be created and assigned to different processors (perhaps using a table which is prepared in advance and used to direct start-up). The initial number of instances of each generic process will be based on an analysis of overall bandwidth and redundancy requirements. Thereafter, the number may be changed from time to time by a background application process which will monitor workloads and will react if capacity is lost because of hardware failure or if certain tasks require more service because of changes in demand. Typically there will not be more than one instance of any process on any one processor. Initially, we may place a copy of every process on every processor, but as we gain experience with the system or as memory becomes scarce, we may want to be more selective.

By performing global scheduling at the point when the system appears most tightly coupled, and by maintaining copies of each process on several processors, we will be able to provide an environment in which load can be easily distributed according to processor availability and in which application processes can efficiently ignore distinctions between processors. Whether we can provide this environment with low overhead and low delay will depend upon the algorithm chosen for assigning specific tasks to specific processes. The choice of an algorithm will be considered in the next four sections. At this time we are

considering a number of options, our intention being to select the simplest one which is based on sound premises and meets the requirements of the Voice Funnel.

Regardless of the scheduling strategy chosen, tasks will be moved from originator to server by primitive functions called SEND and RECEIVE, which will be mentioned frequently in subsequent sections. SEND will submit a task for processing by a server, using whatever technique is dictated by scheduling policy, and RECEIVE will be the means by which an idle server solicits more work.

4.2.1 Criteria

There are a number of criteria by which to judge any global scheduling algorithm for the Butterfly Multiprocessor. Before proceeding with specific algorithms, we will consider the more significant criteria, in approximate order of importance:

- robustness: vulnerability of the system to loss of memory, loss of a processor, or corruption of data structures;
- latency: the time which elapses before a task is started, including the time waiting to be scheduled, plus the time waiting to be run; alternatively, the time waiting while lower priority tasks or younger tasks of equal priority are being run;
- load leveling: the extent to which work can be quickly transferred from processors which are too busy to those which are lightly loaded;
- time required to queue a task for further processing: an important component of latency; also determines the speed

with which processors which are over-loaded can transfer excess load elsewhere;

- time required to decide where a task should be performed: the time spent making this choice is an important component of both latency and system overhead; the quality of this decision determines subsequent latency;

- queue contention: excessive contention for common resources, such as scheduling queues, would significantly increase delay and would waste processor cycles;

- consideration of priority: the extent to which higher priority tasks take precedence over lower priority tasks;

- consideration of age: the extent to which the older of two tasks of equal priority (e.g., two voice parcels) would take precedence over the younger; less stringently, the extent to which the older of two occurrences of the same generic task would take precedence over the younger, ignoring any relationship between the ages of tasks not of the same type, or the extent to which the older of two tasks originating on the same processor takes precedence over the younger;

- overhead: the proportion of total processor bandwidth required for executing scheduling algorithms, time which cannot be directly applied to the application;

- facilities for influencing selection: the extent to which the scheduling algorithm allows the programmer to specify that a task must be assigned to a process on a designated processor, or that it would be desirable (although not necessary) for it to be assigned to a process on a designated processor; the former capability must be provided (for example, when access to a specific device is required), and the latter would be desirable (for example, as an aid to reducing remote references).

4.2.2 Implicit Global Scheduling

An especially straightforward algorithm for global scheduling would be to simply use SEND and RECEIVE. Depending on whether or not the task being scheduled required (or preferred) a

particular processor, the process originating a new task would send it to the required (or preferred) processor or would add it to a global queue for the generic task. The implicit global scheduling option would have the attractive property that global scheduling would occur as a consequence of the natural flow of the system. In effect, there would be no explicit global scheduler, but instead tasks for which scheduling discretion existed would flow naturally toward preferred processors when possible and toward less busy processors otherwise.

This approach would use three basic modules (two functions and an interrupt service routine) and a number of queues and eventcounts (see Section 3.3). The basic modules would be:

- SEND (embedded in each process as a function),
- RECEIVE (embedded in each process as a function), and
- an inter-processor interrupt service routine (one for each processor).

The queues would include:

- one global queue for each generic task for which no particular processor would ever be specified or for which a specified preference might be ignored by the designated processor if it were busy;
- for each processor, one local queue for each generic task for which the processor might be a required or preferred server;
- for each processor, one inter-processor queue for receiving tasks sent from other processors.

The eventcounts would include:

- one for each task queue (to indicate additions to the queue),
- one for each task queue (to indicate removals from the queue),
- one for each task class* (to indicate quickly to all processors that some task in the class had been added to a global queue),
- for each processor, a work-pending eventcount for each priority level, and
- for each processor, a work-finished eventcount for each priority level.

Note that, with respect to processor preference, generic tasks would have one of three types: those for which no processor were preferred, those for which a particular processor were preferred but not required, and those for which a particular processor were required. Tasks of the first type would always be placed on a global queue for service, and those of the third type would always be placed on a service queue local to the required processor. Tasks of the second type would be placed on either a global or a local service queue, depending on whether or not the

* To reduce the number of variables to be examined at clock interrupt time, generic tasks would be grouped into classes by priority and by distribution of servers. For example, one class might consist of all high priority tasks served by encoder nodes, and another might consist of all high priority tasks served by PSAT nodes. Each clock interrupt routine would check only those task class eventcounts for which its node had servers and which were for priorities higher than that of the current process. When it noticed that a task had been added to a higher priority global queue which it served, the clock interrupt routine would invoke the local scheduler, which would pre-empt the current process.

preferred processor were too busy. To avoid requiring processes which served such tasks to monitor both a local service queue and a global one, and to avoid giving either of the two queues preference over the other, any processor which served such a generic task would have distinct servers for the local and global queues.

The following key terms will be used in the following descriptions of the basic modules (terms which are self-explanatory are not listed):

Occurrence - refers to an occurrence of a generic task;

Priority - is an attribute of GenericTask;

ServiceQueue - is the queue from which a process would obtain the tasks it was to perform;

GlobalAcceptLimit(Priority) - the maximum amount by which WorkPending(Priority) could exceed WorkFinished(Priority) before a processor would consider itself so busy that it would stop seeking work from global service queues; the limit might be a constant or it might be a variable that was adjusted periodically by a background process; it might or might not be the same for all processors;

PreferredAcceptLimit(Priority) - the maximum amount by which WorkPending(Priority) could exceed WorkFinished(Priority) before a processor would consider itself so busy that it would stop accepting tasks which were sent to it because it was the preferred server; as with GlobalAcceptLimit, it might be a constant or a variable adjusted by a background process, and it might or might not be the same for all processors;

ADVANCE - would increment an eventcount (See Section 3.3);

AWAIT - would block the calling process until an eventcount reached a specified value;

WAIT - would block the calling process until a specified condition were met;

PUT - would add an element to a queue;

GET - would return the first element from a queue unless the queue were empty, in which case it would return NULL.

Descriptions of the three basic modules will now be presented.

```
SEND (Occurrence, GenericTask, Processor)

     IF GenericTask.Type is REQUIRE or PREFER
          PUT Occurrence in InterProcessorQueue
               for Processor
          interrupt Processor (or invoke local routine if Self)
          RETURN
     OTHERWISE
          PUT Occurrence in global queue for GenericTask
          ADVANCE eventcount for global queue to reflect
               addition
          ADVANCE eventcount for task class (to alert
               clock interrupt routines)
          RETURN
```

[Version of RECEIVE for Local Service Queues]

```
RECEIVE (GenericTask, ServiceQueue) RETURNS(Occurrence)

     ADVANCE WorkFinished(Priority)
     REPEAT FOREVER
          Occurrence = GET(ServiceQueue)
          IF Occurrence not NULL
               ADVANCE eventcount for ServiceQueue
                    to indicate removal
               RETURN (Occurrence)
          ELSE AWAIT ADVANCE of eventcount indicating
               new arrival in ServiceQueue
```

```
[Version of RECEIVE for Global Service Queues]

RECEIVE (GenericTask, ServiceQueue) RETURNS(Occurrence)

     ADVANCE WorkFinished(Priority)
     DISMISS to scheduler (having just finished a task, get a
          fresh time slice before committing to perform a new
          task; may lose control if another process ready)
     REPEAT FOREVER
          IF WorkPending - WorkFinished >= GlobalAcceptLimit
                    (for this Priority)
               WAIT for appropriate reduction of difference
          Occurrence = GET(ServiceQueue)
          IF Occurrence not NULL
               ADVANCE eventcount for ServiceQueue
                    to indicate removal
               ADVANCE WorkPending(Priority)
               RETURN (Occurrence)
          ELSE AWAIT ADVANCE of eventcount indicating
               new arrival in ServiceQueue

INTERRUPT SERVICE ROUTINE FOR INTER-PROCESSOR INTERRUPTS

     REPEAT UNTIL InterProcessorQueue empty
          Occurrence = GET(InterProcessorQueue(Self))
          IF GenericTask.Type is REQUIRE
          OR IF WorkPending - WorkFinished
                    < PreferredAcceptLimit
                    (for Priority of GenericTask)
               PUT Occurrence in local queue
                    for GenericTask
               ADVANCE eventcount for local queue to
                    reflect addition
               ADVANCE WorkPending(Priority)
          OTHERWISE
               PUT Occurrence in global queue for
                    GenericTask
               ADVANCE eventcount for global queue to
                    reflect addition
               ADVANCE eventcount for task class (to
                    alert clock interrupt routines)
     IF any new Occurrences were of higher priority than
          the current process EXIT to local scheduler
     ELSE RETURN from interrupt
```

Note that certain aspects of the proposed algorithm might be modified after further study; for example:

- instead of having SEND interrupt the target processor for tasks for which a particular processor were required or preferred, it might be sufficient to rely on the frequency of other interrupts on the target processor to assure low latency;

- the entire interrupt routine for tasks preferring or requiring a particular processor could be eliminated (and the movement of a task from an inter-processor queue to a local or a global scheduling queue could be avoided) if the appropriate decisions were incorporated into SEND; however, this would move the burden of such effort to the sending processor; in addition, the current algorithm would have the advantage that each processor would have exclusive access to its own local queues and would make its own decisions regarding how busy it was;

- the pair of eventcounts indicating addition and removal for each queue could be eliminated if processes were to wait directly on queues becoming non-empty; whether or not this were done would depend on which implementation were more reliable and more efficient.

Upon inspection, the implicit option can be seen to meet the global scheduling criteria very well:

- with appropriate support from the reliability software, the algorithm should be fairly robust; information would be distributed over a number of queues, many local to individual processors, so loss of a single queue should not be catastrophic; while the algorithm would depend heavily on eventcounts, it would generally be possible for reliability software to reconstruct at least relative relationships between eventcounts from queue lengths; since eventcounts would generally be used by this algorithm only to detect changes, it should be safe to arbitrarily advance selected eventcounts if something were stuck;

- through the use of the inter-processor interrupt facility when sending tasks to specific processors, and through reliance on the clock interrupt routine to check frequently for new occurrences of tasks in global queues, the algorithm should maintain a low level of latency;

- because any busy processor could decline to accept tasks which it was not required to handle, and because all

processors which were not busy would check global queues when they detected that new entries had been made, load leveling should occur naturally;

- because SEND need only queue a task and then either interrupt the target processor or ADVANCE two eventcounts, the time required to queue a task for further processing would be very brief;

- likewise, the time required to decide where to perform a task would be small; tasks for which a particular processor were required or preferred would be sent there directly (without any scan or computation) and would be queued locally if they could be accepted or globally if not (after a very simple test); tasks for which no preference were expressed, or for which preference could not be honored, would be picked up by a processor which would only have to check one global queue which it would know had had a recent entry;

- the use of a large number of processor-specific and task-specific queues would tend to keep contention low; in any event, the most likely source of contention would be the global queues; in evaluating this, we note that for a queue with Poisson arrivals, the average wait engendered by contention will approach the nominal service time (doubling effective service time) as the number of requests approaches half of total capacity, and that the wait will begin to get much worse somewhat thereafter; thus, with a nominal service time of 20usecs (in this case, time to insert or remove a queue element), each global queue could handle up to 25000 requests per second (equivalent to 500 vocoders at 50 parcels per second); for those queues which did approach half of capacity, it would be relatively easy to partition the work to reduce contention;

- priority would be observed because the local schedulers would run servers in priority order whenever there were tasks in their service queues (local or global);

- because tasks would be queued in order of arrival, older tasks would be served ahead of younger tasks when both were placed in the same queue; however, when tasks were placed in different queues (because they were of different generic types or because they were directed to different processors), the order in which they would be serviced would be indeterminate; as long as most queues tended to be empty, this would not create unreasonable delays; if

most queues were not empty most of the time, however, that would probably indicate that the system were overloaded;

- since each of the basic modules would only execute a few statements for each item it would handle, and since loops would only occur in exceptional circumstances, the overhead imposed by the algorithm would be small;

- since the sender could include a preference in the call to SEND, and since the receiver could easily determine whether to honor the preference, it would be very easy to influence processor selection.

4.2.3 Decentralized Global Scheduling

A very different approach from the one just discussed would be to provide an explicit global scheduler on each processor. The function of this global scheduler would be to pull toward each processor tasks for which it had suitable processes, to the extent that the processor had the capacity to take on additional work. By having each processor take work as capacity became available, post-scheduling latency would be reduced to a negligible level. In addition, each processor would run its global scheduler only when it had no more urgent work to perform, and not when it was busy. This approach will be described briefly, but will not be reviewed against the criteria.

Any process which originated a task would send it, as with the previous approach. If the task required a particular processor, SEND would place it in a task-specific queue local to that processor. If there were no preference, or a non-binding preference, SEND would place the task in a global queue for the

generic task. SEND would not attempt to awaken any serving process. Upon entry, RECEIVE would block until awakened by the global scheduling module for its processor, after which it would obtain the next entry from the task queue which was specified when it was called (if another processor managed to empty the queue first, RECEIVE would block again).

A new module, called GLOBAL-SCHEDULER, would be invoked whenever no high priority processes were ready to run. GLOBAL-SCHEDULER would inspect the high priority entries in the queue of processes which were waiting as a result of entering RECEIVE (this queue would be in FIFO order by priority) and would select the first process whose queue of waiting tasks was not empty and make that process ready. If all high priority queues were empty, GLOBAL-SCHEDULER would turn its attention to the next priority level, unless it considered its processor to be too busy at that level to take on additional tasks.

4.2.4 Centralized Global Scheduling

A third approach would utilize a centralized global scheduling module, which might be run on a dedicated processor, or which might be passed from processor to processor. In contrast to the previous algorithm, this algorithm would push work towards those processors which were less busy than others. It, too, will be briefly described but will not be compared against the criteria.

As with the previous approaches, individual processes would submit and obtain new work via SEND and RECEIVE. In this case, however, both SEND and RECEIVE would be extremely simple. SEND would always put new tasks into a single local queue, regardless of type or preference. RECEIVE would block and, upon being awakened, would be given a pointer to a task.

A central module, GLOBAL-SCHEDULER, which would run on only one processor at a time, would scan the local SEND queues regularly. Upon finding an unscheduled task, GLOBAL-SCHEDULER would select a processor which was running at a priority level lower than that of the new task, or failing that, would select a processor which was not too busy. GLOBAL-SCHEDULER would then add the task to the appropriate task input queue for the selected processor, if one existed, and would send the selected processor an interrupt at the level of the task. Tasks which required a specific processor would be assigned to that processor, while tasks which indicated a preference for a particular processor would be assigned to that processor unless it were too busy, in which case preference would be ignored.

Responsibility for global scheduling could be passed around fairly easily. When the GLOBAL-SCHEDULER became active on a particular processor, it could set a global variable to identify itself. When it had run long enough, it could set a second global variable to indicate that it had quit. These global

variables could be inspected by all processors whenever they ran out of high priority work. The first to discover that the last GLOBAL-SCHEDULER had ceased could take over. Together with a watchdog timer, this mechanism would provide a means of detecting the loss of the GLOBAL-SCHEDULER, should that occur.

4.3 Local Scheduler

Each processor will have a local scheduler which will be responsible for the short-term scheduling of the processes which have been assigned to it. The local scheduler will be invoked whenever an active process blocks, whenever a device interrupt causes a higher priority task than the one being processed to be queued for service, and whenever a software timer reaches zero. When invoked, the local scheduler will determine which of its ready processes should run next (based primarily, if not exclusively, on priority), will perform any necessary context switching, and will give control to the new process. The local scheduler will be completely unaware of global scheduling issues or even of the existence of other processors.

We anticipate a small number (currently four) of levels of process priority. Although adding more would be a trivial extension, having too many levels might increase scheduling overhead unnecessarily. These levels have tentatively been allocated as follows:

- level 1: processes which must complete in time for the next PSAT data transfer (for example, processes which collect parcels from many encoders and arrange them in a PSAT buffer);
- level 2: processes which must keep up with encoders (for example, processes which add headers to voice parcels or which play voice parcels back to encoders);
- level 3: processes having more relaxed real-time requirements but still having to respect human expectations (for example, processes which perform call setup); and
- level 4: background processes having no specific real-time requirements other than that they complete within a reasonable time.

In order to minimize scheduling latency, each interrupt service routine (including the clock routine) will check whether a local scheduling event (such as the completion of a data transfer) or a global scheduling event (such as adding a task to a global scheduling queue) has occurred and might make a higher priority process runnable. If so the service routine will give control to the local scheduler rather than returning to the process which was interrupted.

There will be a time slice associated with each priority level. The scheduler will not suspend one process to run another of equal or lower priority until the first has exhausted its time slice. Processes which must run at high priority will be given a small time slice. However, since high priority tasks are expected to be very short, the high priority time slice will be large enough to permit routine processing of a task to complete.

Low priority tasks, which will normally be run when when there are no high priority tasks waiting, will receive longer time slices, while intermediate priority tasks will receive slices of intermediate length. In general, time slices will be assigned to minimize process switching, while ensuring that erroneous processes do not get to run indefinitely and ensuring that background processes get regular intervals of service without excessive switching. Any process which is designed to run continuously, without ever blocking, should occasionally wait for an an appropriate interval of time to elapse, in order not to prevent processes of lower priority from ever getting to run.

Interrupt service routines will have priority over all processes. Whenever an interrupt is granted, control will be given to the appropriate service routine immediately. When the service routine has finished servicing the interrupt, it will give control to the local scheduler which will determine whether to return control to the process which was interrupted or to give control to another process. The extent to which interrupt service routines might be allowed to interrupt one another is yet to be determined and will depend on the facilities provided by the processor and the interrupt logic.

We have not yet decided on a strategy for actually making processes runnable, but are considering two general strategies. The first of these strategies would have the local scheduler,

during its scan at each priority level, perform a simple test (probably a comparison of two objects indicated by pointers) to determine whether a condition specified by a process had been met. The second strategy would have the process which caused the condition to be met (or ascertained that it had been met) move any processes waiting on the condition to their respective local ready queues (using a system call). The latter strategy would allow the local scheduler to react more quickly, but may involve more overall work, especially when several processes were waiting on the same condition. The former strategy would permit each processor to manage its own scheduling queues and would provide for better separation between processors. We prefer to make the final decision on this question after other details of global and local scheduling strategy, together with details of data structures and details of queue management, have been worked out.

5. Memory Management

In this section we consider strategies for utilizing a segmented address space to provide flexibility and protection at reasonable cost. Segmentation, multiple processes, and multiple processors interact to create complications, but by adopting a limited set of goals we will be able to utilize them to our advantage. In the following pages, we discuss the issues which arise and we then examine two possible approaches, one of which provides a fully dynamic scheme at run-time, while the other fixes many things at link time. We conclude that the latter approach provides adequate flexibility and protection, is easier to implement, and is more efficient at run-time.

5.1 Memory Usage Within and Between Processes

A number of issues exist concerning memory usage. Among the more important are:

allowing processes to share memory,
managing free space,
allocating dynamic memory,
permitting inter-processor references,
switching context, and
utilizing a limited range of segment numbers.

The mechanics of sharing data are rather straightforward, requiring mechanisms for specifying what is to be shared and for establishing the appropriate connections. Shared references (as opposed to shared data) do present a problem, however. If two

processes share a pointer* to an item (or share an instruction which references an item), then either (1) the item must be in the address space of both processes and must have the same address** in both address spaces or (2) the reference must be indirect and must be translated into the virtual address which is correct for each process. For the item to have the same virtual address in both address spaces, specific action would have to be taken or constraints would have to be observed, as otherwise different addresses would probably be assigned [BRIN 73]. Indirect references would be quite slow, since translations would have to be performed whenever they were made, and translation tables which might be quite large would need to be maintained for each process.

The existence of free space, which could be managed on behalf of a process to provide dynamic variables, would require a free space manager; but that machinery need not be too complicated.

Given the existence of dynamic variables, it would be attractive to have the ability to dynamically expand and contract individual address spaces, as this would allow each process to

---

* Pointers are full virtual addresses, as opposed to indexes, which are offsets from some base such as the beginning of an array.

** Except when otherwise stated, all addresses mentioned will be virtual addresses.

get the memory it needed without requiring it to hold much more than it was actually using. This could be accomplished either by adding and removing segments or by increasing or shrinking segments already held. While contracting an address space would be straightforward, both methods for expanding one would not. Expanding segments would be difficult because the Z8000 memory is not paged and the application neither permits nor requires swapping. Because the memory is not paged, segments could not be extended in place unless they happened to be adjacent to free space. Because swapping will not be done, natural opportunities to move segments will not arise; and the delay caused by rearranging memory to create adjacent free space, and updating the affected memory maps in each processor, would be prohibitive. Adding segments could be done, but only to the extent that free segment numbers were available. Further, when a new segment was to be shared, if it were not possible to assign it the same number in all address spaces which shared it, it would be necessary to use indirect methods for any shared references to it.

The architecture of the Butterfly Multiprocessor will make references between processors very natural, since the hardware will allow the virtual memory of any process to be mapped (via segmentation registers) into areas of memory in any processor. Although access times may differ, processor boundaries will be

invisible to the programmer. Each process may therefore be written and linked without regard to where it will be loaded. The assignment of physical memory to the segments of the process' virtual address space can be made at load time according to such considerations as where (in which processor) the process will run, which segments must be local to one another, whether segments are to be shared (and are already loaded), and general availability of memory.

While segments provide several benefits, such as convenient sharing, protection , and allocation of memory, they also have their disadvantages. When control of a particular processor passes from one process to another, all processor registers (including memory mapping registers) must be reloaded (context must be switched) to reflect the state (such as the address space) of the new process. The greater the extent to which the address space of the new process differs from that of its predecessor, the greater will be the cost of the context switch (solutions to this problem will be considered in Section 5.2.5).

While some systems (e.g., MULTICS) have had a virtually unlimited range of segment numbers (2**36), it is more common to have a rather limited range (8 for PDP-11, 128 for the Z8000). The range of the Z8000 is probably typical for the processors which might be used for the Butterfly Multiprocessor in the near future. Because this range is limited, and because for

efficiency reasons one might wish to reserve some of the available registers (say 25%) for system use (to avoid the need to load mapping registers at interrupt time), it will be necessary to avoid schemes which require an unlimited number of segments. Thus, for example, the number of segments which a process might share simultaneously will be limited. This in turn will mean that, on a large system, a process cannot have a window into every processor (without changing mapping register contents), nor can it have intimate sharing relationships with a very large number of processes.

5.2 Memory Management: Some Approaches and Tradeoffs

To summarize the previous discussion,

- Memory usage falls into a number of distinct classes, of which most are private to individual processes. Private memory presents no unusual problems. Instructions may be shared or private, but even if shared, they present no problems not encountered with shared data. Shared data may be of two types, static and dynamic.
- Data shared by several processes does present a problem, because it exists in more than one address space. If it does not have the same address in all of those address spaces, then processes may not share direct references (e.g., pointers or instruction references) to it, but must make shared references via a translation table. On the other hand, arranging for shared data to have the same address in all address spaces to which it belongs is difficult, because the asynchronous expansion and contraction of various address spaces would quickly get out of phase [BRIN 73].
- Dynamic allocation of data within the current address space requires the provision of a free space manager. No significant problems arise unless the free space manager must be able to expand the virtual address space of the process.

- Address spaces are hard to expand, especially where shared data are concerned, because segments cannot usually be expanded and because it is hard to avoid assigning different numbers when a segment is added to more than one address space.

- Inter-processor references will occur very naturally whenever the loader arranges the segment map for an address space to include physical memory from more than one processor; neither the programmer nor the linker need be concerned with this.

- While segments provide a useful protection mechanism, it is desirable to minimize the number of segments which differ from process to process, in order to keep context switching costs as low as possible.

- A second reason for conserving segments is that the total number of segments will be limited in most near term hardware implementations of the Butterfly Multiprocessor. For example, the Z8000 has only 128 segments, of which (say) 32 may be reserved for system use.

Two general approaches will now be considered in light of the above. The first uses indirect pointers for references to shared segments. It is more flexible than the second (or at least more dynamic), but is harder to manage and requires more processing at run time. The second requires segment numbers to be fixed at link time. It gives up some of the flexibility of the first approach in order to gain greater run time efficiency. We recommend the second approach.

5.2.1 Dynamic Segment Approach

Under this approach, both the linker and the run time memory manager would place new segments wherever a free segment number existed in the address space of the process being served, using

whatever selection criteria were desired. There would, in general, be no need to consider the segment's position in the address space of other processes.

This approach would work perfectly well for segments which were known only to one process; no special arrangements would be required. Static data in shared segments could also be referenced directly by instructions (provided the instructions were not shared), and both static and dynamic data in shared segments could be referenced directly by private pointers. However, shared pointers would not, in general, work correctly, since each process might consider the datum to be located in a different segment within its own address space.

A possible mechanism for coping with this problem would be to store, not a pointer, but a pair of items <index,offset>, when storing a shared reference. "Offset" would be merely the offset part of a normal pointer, while "index" would be a globally unique index into a table of shared segments (each shared segment would have a distinct entry). Each process would contain a system-created table in its address space which would contain its own segment numbers for each of the segments it shared. Using the notation of the C Programming Language, instead of using a pointer (*P), one would use a pointer expression (*(T(I)+O), where T would be a table of segment numbers, I an index, and O an offset). It is possible that the <index,offset> construction

could be added to C as a new built-in data type, in which case it would be easy to use and would perhaps require only one extra memory reference. Without such assistance, this mechanism would be somewhat error prone and probably more expensive to execute. Two further drawbacks would be the machinery required to maintain the global and private copies of the mapping table, and the need to store a table in the address space of every process with shared segments (note that because each process would reference this table with a globally unique logical segment number, the range of indexes, and hence the size of the table, could be very large).

5.2.2 Fixed Segment Approach

One might instead require that the numbers for all shared segments be assigned and fixed at link time. This might be done conveniently by having a single link step (see Section 7.4) in which all shared segments for a particular application were assigned and recorded for later use in linking the individual processes. It would not be too difficult to use a tree structure (like a traditional overlay structure) for this, so that groups of processes with disjoint sharing requirements could each use a block of segmerts for its own purposes, while perhaps having another block in common. Different applications could each have their own sharing trees.

This approach would have the advantage of simplicity, in that a uniform method could be used for referring to both private and shared data, and in that there would be no need to maintain a translation table for each process. The disadvantage of this approach would be that the size of all shared data areas would be fixed at link time, which would mean that they could not start out small and grow to accommodate need (some provision would also be necessary for segments which would take "what was left", such as buffer pools).

As with the previous approach, it would be necessary to maintain a global table mapping the names of shared segments into physical memory locations. However, it would not be necessary for every process to have a copy. The loader would use and maintain this table when loading new processes.

5.2.3 Free Space Management

Free space might be made available in small units to a process (if private) or processes (if shared) from a larger area obtained at run time (first approach) or at link time (second approach). (Actually, one might choose approach one for private areas and approach two for shared areas.) For convenience, each process might be allowed multiple free space areas (perhaps for allocation units of a particular size or duration), to facilitate storage management and garbage collection. Similarly, there

might be a number of shared free space areas. The various areas would be identified by private or shared names, as appropriate.

Free space areas acquired at run time (first approach) might possibly be expanded by allowing an area to have non-contiguous extents (assuming all processes having access to an area being expanded had a segment number available). Free space areas acquired at link time (second approach) would not be expandable.

5.2.4 Conserving Segment Numbers

With either of the two schemes proposed above, various steps could be taken to conserve segment numbers. In general, the instructions, private static data , and stack for a particular process could each be assigned a single number (the latter two might also be combined). In addition, dynamic areas having similar attributes (protection, whether shared, and by whom shared) could be consolidated, as could shared static data having similar attributes.

5.2.5 Reducing the Context to be Switched

By allowing segments to be assigned on a process by process basis, the first approach would tend to place shared segments in different places in each memory map. The second approach would always assign the same segment number to shared segments, which would at least make it possible to consider not reloading the

might be a number of shared free space areas. The various areas would be identified by private or shared names, as appropriate.

Free space areas acquired at run time (first approach) might possibly be expanded by allowing an area to have non-contiguous extents (assuming all processes having access to an area being expanded had a segment number available). Free space areas acquired at link time (second approach) would not be expandable.

5.2.4 Conserving Segment Numbers

With either of the two schemes proposed above, various steps could be taken to conserve segment numbers. In general, the instructions, private static data , and stack for a particular process could each be assigned a single number (the latter two might also be combined). In addition, dynamic areas having similar attributes (protection, whether shared, and by whom shared) could be consolidated, as could shared static data having similar attributes.

5.2.5 Reducing the Context to be Switched

By allowing segments to be assigned on a process by process basis, the first approach would tend to place shared segments in different places in each memory map. The second approach would always assign the same segment number to shared segments, which would at least make it possible to consider not reloading the

mapping registers for segments shared by the new and old processes.

To reduce the amount of context which must be switched when the local scheduler stops one process and permits another to run, we have defined an entity called a domain (see Section 7.4). A domain will consist of a group of segments which will be treated as a unit for sharing purposes (i.e., sharing relationships will be declared in terms of domains rather than segments). When context is switched, it will then only be necessary to compare domain lists (which can be very much shorter than segment maps) in order to determine what changes to make to the segment registers. The programmer may use domains to make tradeoffs between protection (many domains having few segments each and not widely shared) and context switching efficiency (few widely shared domains). In fact, the programmer who wishes to minimize context switching might include all instruction segments for all application processes on a node in a single shared domain and all shared data in another domain, leaving only the private data and the stack (which might share a segment for each process) to be switched.

6. Reliability and Availability

The inherent redundancy of the Butterfly Multiprocessor makes it attractive to consider strategies for providing a high degree of reliability and availability during Voice Funnel operation. The goals of this effort are:

- to continue operation despite the occurrence of most types of single failure;
- to continue operation while a section of the hardware or software is being repaired or replaced;
- to detect most failures within a short time after their occurrence; and
- to contain failures to the locality of the failing component.

The ideas presented in this section are based extensively on those successfully employed in the Pluribus Multiprocessor [ROBI 78, KATS 78], but are modified to reflect differences in the hardware and software architectures of the Pluribus and the Butterfly Multiprocessor. Key among these differences are that:

- I/O devices are common on the Pluribus, but each device will be visible to only one processor on the Butterfly system;
- memory is either common or local on the Pluribus (but not both), while it will always be local to one processor on the Butterfly Multiprocessor and common (remote) to all others;
- Pluribus memory is divided into pages of the same length, and the software is aware of pages, while Butterfly Multiprocessor memory will be divided into segments of different lengths of which most of the software will have no awareness;

- Pluribus software is divided into strips whose length is constrained by device latency requirements, while Butterfly Multiprocessor software will be divided into processes which will have no such constraints; and
- Pluribus software is written in assembly language, while Butterfly Multiprocessor software will be written in C.

Among the ideas which the Butterfly Multiprocessor operating system will borrow from the Pluribus are:

- discovery and verification of the hardware configuration at system initialization time;
- development and maintenance of a consensus regarding the set of correctly functioning common resources;
- periodic rechecking of critical resources;
- use of redundant information for verification of data structures;
- timeouts for locks on common resources; and
- blocking of higher level activities when lower level activities have detected possible errors or inconsistencies in resources upon which the higher level activities depend.

Some of the more important aspects of the reliability strategy will be discussed in more detail in the following sections. The ideas presented here are tentative and are meant primarily as an indication of the approach we will take. As a general rule, we will look closely at Pluribus experience and solutions before making firm decisions regarding strategy in a particular area.

6.1 Organization of the Reliability Software

Each primary component in the operating system and the application (e.g., scheduler, memory manager, or communications manager) will have a service portion, a reliability portion, and a fix-up portion. The service portion will provide the services which the component was specified to offer, the reliability portion will provide knowledgeable verification of the entire component, and the fix-up portion will endeavor to put things right when there is a consensus that an error exists. This organization makes it unnecessary to have a monolithic reliability module which has knowledge of, and access to, all code and data structures. Instead, the reliability module need only know which component-specific functions to call to perform periodic checks of each component and to fix things up when there is a consensus that they are out of order.

Reliability software and fix-up software will be run under any of the following circumstances:

- at system startup;
- periodically during normal operation; and
- when there has been an error of appropriate severity.

During normal operation, a number of reliability processes (corresponding to various levels of confidence in system integrity, cf. Pluribus Stages) will run in the background on

each processor. In addition, the timer service routine for each processor will check certain watchdog timers, some of which will be used to ensure that the reliability processes receive at least a minimum allotment of time, while others will be used to check for such things as locks or synchronizing variables which have become stuck. If the reliability processes are not being run often enough, the timer service routine will move them to successively higher priority levels until they get enough time. If the timer service routine has reason to suspect serious problems, it may even inhibit interrupts and give full control to the reliability processes.

Each reliability process (including timers) will have a flag (or synchronizing variable) associated with it which will indicate whether it considers everything to be all right. Whenever it has completed its tests successfully, it will set its flag true; and whenever it finds a problem, it will set its own flag false, together with all flags associated with any reliability processes which depend on it. In addition, each reliability process will block itself if any of the flags for components (possibly more than one) upon which it depends have become false. Finally, the scheduler will not run normal processes unless a composite reliability flag is set. (This general mechanism is very similar to the Pluribus Stage mechanism, except that here any particular component can depend

on more than one lower level component, and scheduling is slightly different.)

Because reliability processes might run for a long time, they may be time-sliced by the scheduler so that other processes can run. If a reliability process needs exclusive access to a data structure, it must lock the structure, and will have its priority promoted to that of the lock (determined by the highest priority of any process which needs access to the lock) for the duration of the locking operation.

6.2 System Initialization

When the system is loaded, the processor connected to the Down Line Controller (DLC, see Section 7) will temporarily be master. It will first attempt to verify the integrity of its own memory and processor and the interface to the DLC, following which it will either proceed or ask to be replaced. Upon proceeding, it will install a memory tester in every other node and start it. The memory testers will attempt to determine the state of the local memory for each node and will report back to the temporary master. When all other nodes have either reported back or timed out, the master will initiate some amount of checking among nodes (each node's assessment of its own memory will be checked by one or more other nodes).

Once a consensus has been reached regarding the availability of memory, the master will proceed with loading (as described in Section 7.5), leading to a determination of the full configuration. If any processor determines that it is functioning incorrectly, or if at any time it disagrees about the current environment, it will be removed from the system. No attempt will be made to reach a consensus on devices, since each device is connected to only one processor (some interfaces may be doubly connected, in which case two processor nodes must agree which is to drive the interface).

6.3 Memory Verification During Normal Operation

During normal operation, memory verification will be done differently for code and data. Each processor will have a reliability process which will periodically checksum the instruction segments in each of its processes. This activity will be facilitated by the linker or the loader, which will place code checksums at the beginning of each instruction segment. This checking will be local to each processor. Data segments cannot be checked by such a simple, uniform procedure. Instead, they will be checked by the reliability portion of each component which cares about them. Such checks may or may not be complete, and will depend on the degree of checking required and the degree of redundancy in the data structures.

Redundant copies of data structures which are critical to system operation will be kept on separate processors. To facilitate recovery should the primary copy be lost, the system will have access to a list of the shared segments for each process, indicating the system name for the segment and its position in the address space of the process. The system will maintain (redundantly) a global list of all shared segments and their locations in physical memory. Whenever it is necessary to relocate a shared segment, the memory manager in each processor will be notified and will be able to update any segment maps under its control which may have been affected.

6.4 Timeouts

Watchdog timers will be used to detect a number of conditions, such as:

- reliability processes which are not being run often enough;
- processors which become hung waiting for a spin lock which is not being released; and
- processes which become delayed indefinitely waiting for a synchronization variable which is not being set (presumably because of a failure on the part of another process).

The solution for the first case has already been discussed. In the latter two cases, a strategy similar to that employed in the Pluribus software will be used: when the timeout routine observes that the lock or variable is not being changed, it will unilaterally reset it and will note the occurrence.

7. Development Environment

Our experience with the Pluribus has led us to believe very strongly that the quality of the environment in which software is developed has considerable leverage upon the quality and the cost of the resulting product. With this experience in mind, we have identified a set of mutually complementary facilities which can be provided at a cost which we expect will be substantially less than the savings which result from their availability (many of the facilities already exist and require only slight modification).

The major programs in the development environment will be:

Compiler,
Assembler,
Linker,
Loader (runs on the Butterfly Multiprocessor),
Down Line Controller,
Debugger, and
Various Text Processing Programs.

We intend to use the UNIX operating system and its associated software as the base for our development environment. Many of the programs required are already available under UNIX, and others can be converted to meet our needs. Because it will not be necessary to use any of the UNIX proprietary software in the Voice Funnel itself, we anticipate no licensing problems.

7.1 Program Representation

The system will be partitioned along modular lines roughly corresponding to functional responsibility. The representation of the system will parallel this modular structure, with any necessary interaction between modules corresponding to an externally specified and carefully maintained "uses" hierarchy [PARN 76].

Each part of the system will consist of four items:

module specification,
implementation documentation,
sources, and
test sets.

The module specification will be an implementation-independent statement of the services provided, while the implementation documentation will discuss the details of the design and of the source code.

The entire program representation will be kept on-line in text form. It will be maintained using various programs available in the development environment, such as editors, manuscript processing programs, and the like. No new software development will be necessary for these purposes.

7.2 Programming Language

It is now generally considered that high level languages have evolved to the point where they are competitive with assembly language. Corbato [CORB 72] has stated that experience with MULTICS has confirmed their choice of PL/I as a system programming language (only about six of 1500 modules were recoded in assembly language to achieve efficiency, while several were recoded in PL/I to increase maintainability). Ritchie and Thompson [RITC 78] note that when UNIX was rewritten in C, it grew by about one-third. However, it became much easier to understand and modify, and also included significant functional improvements. Overall, they consider the increase in size to have been "quite acceptable".

With these factors in mind, we have selected C as the programming language for development of the Voice Funnel. We considered a number of other possible choices (ADA, PASCAL, COL, and BLISS among others), and we selected C as the best current compromise between language features and compiler and support program availability. We would prefer to have chosen ADA, since it will be a DOD standard, but did not do so because stable, production quality compilers would not be available in time to ensure completion of the project.

The conversion of the C compiler to produce code for the Z8000 processor on the Voice Funnel will involve converting the code generators of the standard UNIX compiler. Since these stages are largely table driven, this task can be accomplished without significant investment.

None of the design work we have done thus far will require that we add special language features to the compiler. Once the system is operational and stable, however, we may consider converting certain time-critical functions into in-line functions for efficiency purposes.

7.3 Assembler

An assembler will be required for a number of reasons: for translating C output to machine language, for writing machine dependent parts of the operating system, for writing certain diagnostics, and possibly for writing especially time-critical sections of the operating system or the application. Using tools already available to us, we will create a very simple assembler having only the capabilities necessary for supporting the compiler (no macros, no complex expressions, etc.). Previous experience with similar tasks indicates that the effort required will be minimal.

7.4 Linker

The linker will be based on the UNIX linker, which will be modified as necessary. The basic output unit of the linker will be a virtual address space representing one or more processes. Input to the linker will be standard object modules generated by the compiler and the assembler, together with programmer supplied information about segments shared by more than one process. Output will be a set of segments ready to be loaded. Each segment will have various attributes associated with it, such as whether the segment is shared or private and whether it is a member of a shared domain.

To facilitate the management of shared segments, at link time, load time, or run time, it will be possible to assign a name to a block of one or more segments (called a domain) which are to be treated as a unit for sharing purposes. Segment numbers will be assigned to each domain in a link step which must be run whenever the definitions of the domains change. Input to the step will be a tree similar to an overlay tree, consisting of the names of shared variables and structures, their respective protection attributes, the domains into which they are gathered, and a specification of which domains will never occur in the same address space and can therefore be assigned conflicting segment numbers. Output from this step will be a file which will be used when linking processes.

The linker for the Butterfly Multiprocessor will differ from the UNIX linker in knowing about longer addresses (more than 16 bits), in knowing about segments, and in accepting and passing on to the loader information about shared segments and domains. Since the linker will build a virtual address space, it will not need to know about multiple processors. Instead, it will be the loader which maps segments into processors at run time.

7.5 Loader

Loading will be performed by two cooperating sets of programs, one set running on the Butterfly Multiprocessor and one (probably a single program) running on UNIX. This section describes the programs which will run on the Butterfly Multiprocessor; the UNIX program, called the Down Line Controller (DLC), is described in Section 7.9.

System loading will occur in three phases. The first phase will be performed by a PROM-resident bootstrap which will load the second phase over a serial line connected to the DLC and pass control to it. The second phase will verify that it has been loaded correctly, and it will then load a basic operating system (the third phase) into each processor and start it. The third phase will discover the configuration of the processor on which it is running and will assess the hardware. It will then obtain via the DLC any additional processes required for its

configuration and will load and start them. The second and third phases will obtain all necessary files from the DLC using a reliable transmission protocol.

The phase three loaders in the various processors will be supported by a single process (the DLC interface) in the node connected to the DLC. This process will support a link to the DLC (dedicated, dial-up, or network) and will multiplex input and output flowing between the loaders in individual nodes and the DLC.

Once the operating system has been loaded, the primary application process will be loaded. This process will use the facilities provided by the DLC interface and the phase three loaders to load an initial complement of application processes sufficient to handle anticipated load.

7.6 Debugger

Debugging support for the Voice Funnel will consist of a moderately intelligent program running under UNIX (the Down Line Controller) and a cooperating set of very simple programs running in the Voice Funnel. The UNIX program will interact with the programmer and will translate his debugging commands into simple commands (read or modify word, start or stop processor, set breakpoint, etc.) which will be executed locally by the Voice Funnel. A multiplexing module in the Voice Funnel will enable

the UNIX program to interact with a number of local debugging modules in different processors simultaneously. An example of this type of debugger, called XNET, is already running on the ARPANET [MADE 74, TOML 76]; a UNIX version is under development and we hope to be able to use it.

Our experience with the Pluribus, which has had very primitive facilities, has convinced us that we will save substantial amounts of effort later if we apply a small amount early to build reasonable debugging tools. Experience with multiprocessors has also led us to be somewhat wary of the complexities, however, so our goals here are limited.

7.7 Simulator

While we do not currently plan to build a simulator for the Z8000, we may find that such a step is necessary if the early versions of the Z8000 prove undependable.

Nevertheless, because we will be writing most of the software in C and compiling it on UNIX, we will be able to do much of our original testing by first compiling modules to PDP-11 code and testing them under UNIX. This will enable us to perform a significant amount of testing before the first Butterfly Multiprocessor nodes are available, as well as generally permitting us to do early testing of any module in a more easily accessible environment.

7.8 Hardware Facilities for Development Support

Each processor node will have a bootstrap PROM, which will support system loading and provide a set of low level diagnostics and a virtual console. A small number of nodes will be connected to the development machine by serial lines. A given line will provide access from the Down Line Controller in the development machine to the PROM-resident console routines of a particular node.

7.9 Down Line Controller (DLC)

We plan to build a Down Line Controller which will run on the UNIX system. The DLC will be connected to one of the processor nodes on the Butterfly Multiprocessor (via a dedicated, dial-up, or network link) and will have two functions: it will be able to transfer files to a load program running in the Butterfly Multiprocessor, and it will provide an intelligent interface to debugging and diagnostic modules running in the Butterfly Multiprocessor. XNET [TOML 76], which was mentioned earlier, has capabilities similar to those we require.

The load function will be very simple. Programs such as the Butterfly Multiprocessor loader will be able to ask for specific files (typically, but not necessarily, load files created by the linker), and the DLC will read them from the UNIX file system and transmit them to the Butterfly Multiprocessor using some form of

reliable transmission protocol. The DLC will also be able to read information sent by the Butterfly Multiprocessor and save it in a file.

The DLC will also function as an intelligent console for the Butterfly Multiprocessor. It will be the primary means for issuing commands to start, stop or single-step a processor, or cause a bootstrap load. In addition, the DLC will cooperate with the primitive debugging modules in each process to examine or modify memory or to set breakpoints as directed by the programmer. Depending on facilities available under UNIX (such as the ability to create symbol tables), the DLC will be able to provide a modest yet powerful debugging facility.

We have recently used the Down Line Controller technique for development of BBN's Microprogrammable Building Block (MBB). The software took little time to develop and was invaluable in testing and debugging the MBB.

8. References

[BAER 73] J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing", Computer Surveys, Vol. 5, No. 1, March 1973.

[BELL 78] C. G. Bell, J. C. Mudge, and J. E. McNamara, Computer Engineering, Digital Press, 1978, p. 479.

[BRIN 73] P. Brinch Hansen, Operating System Principles, Prentice-Hall, 1973.

[CORB 72] F. J. Corbato, J. H. Saltzer, and C. T. Clingen, "Multics -- The first seven years", AFIPS Conference Proceedings 40, 1972.

[DENN 70] P. J. Denning, "Virtual Memory", Computing Surveys, Vol. 2, No. 3, September 1970.

[DIJK 65] E. W. Dijkstra, "Cooperating Sequential Processes", Technological University Eindhoven, The Netherlands, 1965. (Reprinted in Programming Languages, F. Genuys, ed., Academic Press, 1968.)

[DIJK 68] E. W. Dijkstra, "The Structure of the "THE"-Multiprogramming System", CACM, Vol. 11, No. 5, May 1968.

[ENSL 77] P. H. Enslow Jr., "Multiprocessor Organization -- A Survey", Computing Surveys, Vol. 9, No. 1, March 1977.

[HAVE 78] J. F. Haverty and R. D. Rettberg, "Inter-process Communications for a Server in UNIX", Proceedings of COMPCON 78, 1978.

[KATS 78] D. Katsuki, E. S. Elsam, W. F. Mann, E. S. Roberts, J. G. Robinson, F. S. Skowronski, and E. W. Wolf, "Pluribus -- An Operational Fault-Tolerant Multiprocessor", Proceedings of the IEEE, Vol. 66, No. 10, October 1978.

[KNUE 76] P. Knueven, P. G. Hibbard, and B. W. Leverett, "A Language System for a Multiprocessor Environment", Proceedings of the Fourth International Conference on the Design and Implementation of Algorithmic Languages, Courant Institute of Mathematical Sciences, Computer Sciences Department, New York University, June 1976.

[MADE 74] E. Mader, "Network Debugging Protocol", Request for Comments #643, Bolt Beranek and Newman Inc., July 1974.

[PARN 76] D. L. Parnas, "Some Hypotheses about the "Uses" Hierarchy for Operating Systems", Darmstadt Technical University, Research Report BS I 76/1, September 1976.

[REED 78] D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequencers", CACM, Vol. 22, No. 2, February 1979.

[RITC 78] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978 (revision of July 1974 CACM article, Vol. 17, No. 7).

[ROBI 78] J. G. Robinson and E. S. Roberts, "Software Fault-Tolerance in the Pluribus", AFIPS Conference Proceedings 47, 1978.

[THOR 64] J. E. Thorton, "Parallel Operation in the Control Data 6600", AFIPS Conference Proceedings 26, 1964.

[TOML 76] R. S. Tomlinson, XNET Cross-net Debugger for TENEX User's Manual, BBN Report No. 3377, September 1976.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center (12)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office
R. Brooks
P. Carvey
P. Castleman
W. Clark
F. Heart
M. Hoffman
D. Hunt
B. Hyde
M. Kraley
W. Mann
R. Rettberg
E. Starr
D. Walden
E. Wolf